OSGi 框架概览

1 简介

OSGi 联盟成立于 1999 年 3 月。他的任务是为托管服务从网络到本地网络和设备传输创建开放的规范。OSGi 组织是下一代家庭、汽车、移动电话、桌面系统、移动办公和其它环境的互联网服务的最主要的标准。

OSGi 服务平台规范为服务提供商、开发人员、软件提供商、网关提供了一个开放的、通用体系结构，以协调合作方式来开发、部署和管理服务。它的灵活性和服务的托管部署使得各种智能设备作为一个整体。OSGi 规范面向机器顶盒、服务网关、电缆网卡、电子消费品、PC、工业计算机、汽车、移动电话等。实现 OSGi 规范的设备可以通过网络提供服务。

这是 OSGi 服务平台规范的第四版本，由 OSGi 成员提供。OSGi R4 将大部分 API 扩展到新的应用领域。对已有 API 的更改将保留向前兼容，因此使用前一版本开发的应用系统可以不做任何改动运行在 R4 框架。如果需要的话，内建的版本管理机制允许新的 Bundle 运行在旧的框架。

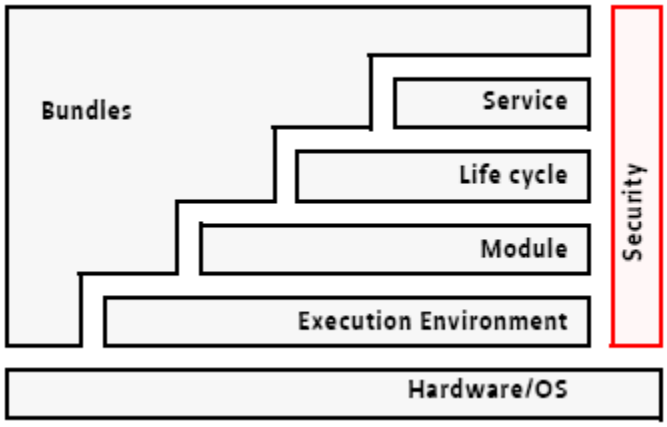
1.1 OSGi 框架概述

Framework 是 OSGi 规范的核心。它提供了一个通用的、安全的和可扩管的 Java 框架，这个框架支持可动态部署和下载的应用程序 Bundle。

OSGi 兼容设置可以下载和安装 OSGi Bundle，并且可以在无需使用它们时删除。这个框架以一种动态和可升级的方式在一个 OSGi 环境中管理 Bundle 的安装和更新。为达到这个目的，它细致的管理了 Bundle 和服务间的依赖关系。

框架为 Bundle 开发人员提供了利用 Java 平台独立和动态代码加载需要的资源，使得能够更加容易在小内存的设备上开发服务且能够大规模部署。

框架的功能被分成以下层次：安全层、模块层、生命周期层、服务层和实际的服务。如下图所示。



安全层基于 Java2 安全，但添加了大量的限制并补充了了 Java 留下的标准。他定义了一个安全包的格式，同时定义了运行时与 Java2 安全层的交互方式。

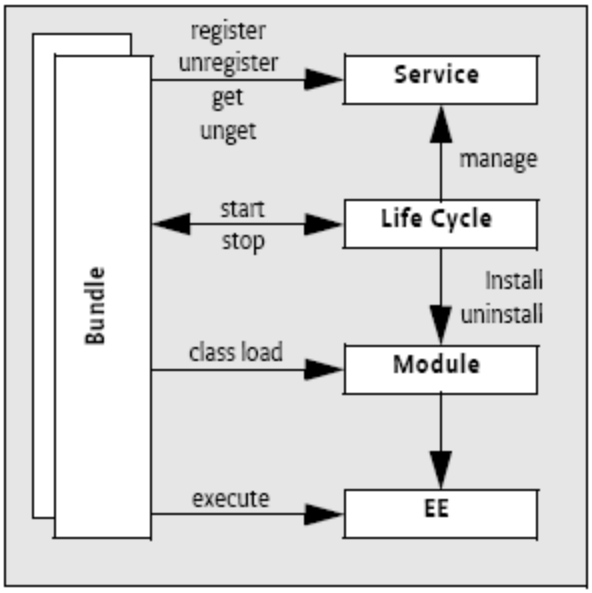
模块层为 Java 定义了模块化模型。他克服了 Java 部署模型的一些缺点。模块化层为 Bundle 间的 Java 包共享或屏蔽共享提供了严格的规则。模块层可以独立于生命周期层和服务层使用。生命周期层提供了管理模块层 Bundle 的 API，而服务层提供了 Bundle 间的通讯模型。

生命周期层提供了 Bundle 的生命周期管理 API。这个 API 为 Bundle 提供了运行时模型。它定义了 Bundle 如何被启动和停止，也定义了如何安装、更新和卸载。此外，它提供了一个完善的事件 API，运行一个管理 Bundle 来控制服务平台的操作。生命周期层依赖于模块层，但可以不依赖安全层。

服务层为 Java Bundle 开发人员提供了一个动态的、简单的和一致的编程模型，它通过将服务规范和服务实现解耦来简化了服务 Bundle 的开发和部署。这个模型允许 Bundle 开发人员来使用服务接口规范绑定到服务。服务特定实现的选择、特定需求的优化或指定的服务提供商，可以延期到运行时决定。

一个一致的编程模型帮助 Bundle 开发人员处理很多不同任务的伸缩性，这是关键的，因为 Framework 是想运行在各种设备，它们有不同的硬件特征，这将影响一个服务平台的各个方面。一致的接口确保软件组件可以被混合、匹配并仍是一个稳定的系统。

Framework 允许 Bundle 在运行时通过 Framework 的服务注册表选择一个合适的实现。Bundle 将根据当前设备的功能注册新的服务、接收服务状态通知或查询已有的服务。Framework 这方面功能使得一个安装后的 Bundle 在部署之后可以被扩展：为新的功能安装新的 Bundle 或在不要求重新系统下更改和更新已有的 Bundle。Framework 层之间的交互如下图所示。



1.2 读者层次

这个规范适合以下读者阅读：

●应用系统开发人员

●框架和系统服务开发人员（系统开发人员）

●架构师

OSGi 规范要求读者至少有一年的 Java 编码实践经验。最好有嵌入式系统和服务经验。应用系统开发人员必须知道 OSGi 环境与传统桌面或服务器环境意义更加重大。

系统开发人员要求对 Java 深入理解。它推荐这些人必须在一个系统环境中至少有 3 年 Java 编码经验。使用 Java 领域的一个 Framework 的实现与传统应用系统实现是不同的。必须详细理解类加载、垃圾收集、Java2 安全和 Java 本地库的装载。

架构师必须关注介绍中的每一个主题。这些介绍包含了这些主题的概述，影响设计的需求、操作的简要描述和使用的实体描述等。这些介绍章节要求有 Java 类和接口的概念，但不要求代码经验。

这些规范大部分适用于应用开发人员和系统开发人员。

1.3 规则和术语

1.3.1 排版

当一个例子包含了一行必须被分成多行是，我们使用‘<<’字符。在这种情况，空格应该被忽略。比如：

http://www.acme.com/sp/ <<

file?abc=12

与

http://www.acme.com/sp/file?abc=12

是等价的。

1.3.2 通用语法定义

在这些规范的很多地方，需要描述一下语法。这个语法基于以下符号。

*	元素重复一次到多次，如（’,’,element）*。
+	重复一次以上。
?	元素是可选的，即出现 0 次或 1 次。
,(…,)	分组。
‘…’	文字。
|	或。
[…]	列表的集合，如 1…5 是 1 2 3 4 5 列表。
<…>	外部定义的符号。

本文预定义了以下符号并在全文中使用。

digit	::= [0…9]
alpha	::= [a…zA…Z]
alphanum	::= alpha | digit
token	::= (alphanum | ‘_’ | ‘-’)+
jletter	::= <查看 JavaLetter 的词汇层次结构>
jletterordigit	::= <同上>
qname	::= <同上>
identifier	::= jletter jletterordigit *
quoted-string	::= ‘”’ ([^ “\#x0D#x0A#x00] | ‘\”’ | ‘\\’)* ‘”’
argument	::= token | quoted-string

directive ::= token ‘:=’ argument （指令）

attribute ::= token ‘=’ argument （特性）

parameter ::= directive | attribute （参数）

unique-name ::= identifier (‘.’ Identifier)* （唯一标识）

symbolic-name::= token(‘.’ token)* （特征名称）

package-name ::= unique-name

path ::= path-unquoted | (‘”’ path-unquoted ‘”’)

path-unquoted ::= path-sep | path –sep ? path-element

(path-sep path-element)*

path-element ::= [^/”\#x0D#x0A#x00]+

path-sep::= ‘/’

前面定义的空白仅当值是可引用的才允许。

Character 类的 isJavadentifierStart 和 isJavaIdentifierPart 方法在最小执行环境中没有定义。这对遵守该规范的嵌入式设备有点困难或代价比较大。因此，允许任何有一个\uoo 的编码一个字符是一个 jleterordigit 或 jletter。

任何包含空白、逗号、小分号、分号、等号或任何不是语法字符的都必须是可打印可引用的字符。

1.3.3 OO 术语

类、接口、对象和服务是截然不同但不好区分的。比如，“LogService”可以表示 LogService 类的一个实例，可以表示 LogService 类，或者表示整个 LogService 的功能。专家通常会从上下文来理解其意义，但这种理解需要付出努力。为了突出这些不同，我们使用以下规则。

当使用类时，它的名称必须与 Java 源文件名一致，使用固定宽度，比如，“HttpService 类”，“在 HttpContext 类的一个方法”或“一个 javax.servlet.Servlet 对象”。当从上下文中无法清楚获取包也无法确定一个类在已知的 Java 包（如 java.lang, java.io, java.util 和 java.net），一个类名必须使用全名格式，比如 javax.servlet.Servlet。否则，包名可以被省略，如 String。

异常和权限类不能跟一个单词“object”。提高可读性必须避免使用“object”后缀。比如“to throw a Security Exception”和“to have File Permission”比“to have a FilePermission object”更具有可读性。

Permission 可以跟上他们的 Actions。ServicePermission[com.acme.*, GET| REGISTER]意味着一个为 com.acme 开头的服务使用 GET 或 REGISTER 动作的 ServicePermission。一个 ServicePermission[Producer|Consumer, REGISTER]意味着这个服务权限对 Producer 和 Consumer 类使用 REGISTER 动作。

当讨论一个类的功能而不是其具体实现时，类名写成普通文本。这个规则在讨论服务时经常使用。比如“the User Admin Service”更具有可读性。

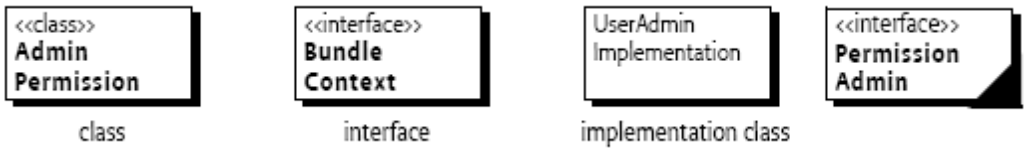
一些服务在类名中嵌入了“Service”。在这种情况下，单词“Service”仅能使用一次且第一个字母必须大写。比如，“the LogService performs”。

服务对象注册到 OSGi 框架。服务注册必须包括服务对象、属性和一系列类和服务实现的接口。这个类和接口必须具有类型安全和命名安全。因此，应该说是一个服务对象是作为一个类/接口注册的。比如，“This service object is registered *under* PermissionAdmin”。

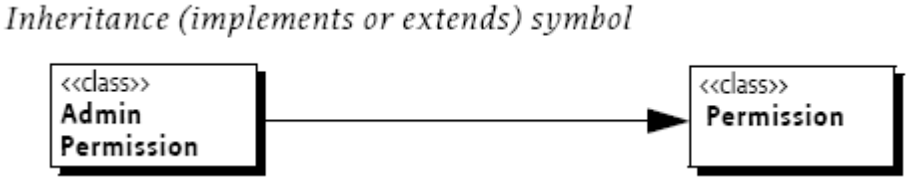
1.3.4 图表

文档的图表示为了说明规范的且不是规范化的。他们的目的是为某一个页提供一个高层概览。以下段落描述了这些图使用的符号和规则。

类和接口被描述为矩形，下图所示。接口在第一行使用<<interface>>标识。规范的类和接口的名称使用粗体。实现类有时候表示为可能的实现。它使用普通字体显示。在有些情况下，类名是省略的，它使用省略号表示。如果一个接口或类作为服务使用，它必须在右下角有一个黑色的三角形。

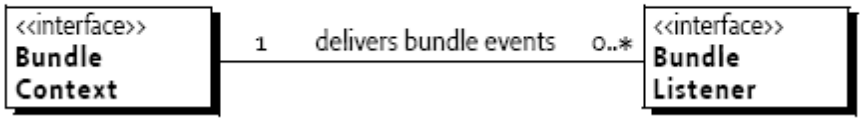


继承、实现或扩展使用一个箭头。



关系使用直线描述。关系的集合使用显眼的标记。下图显示了每一个 BundleContext 对象与 0 到多个的 BundleListener 对象关联，且每一个 BundleListener 仅与一个 BundleContext 关联。关系通常使用一些描述信息。这些描述信息必须是从左到右、从上到下的，且包含了两边的类。比如“一个 BundleContext 对象为 0 到多个 BundleListener 对象提供事件绑定。”

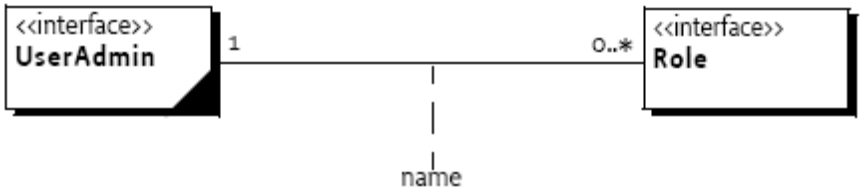
Relations symbol



关联使用一个虚线。关联是类之间存在的且可以存在一个关系中。比如，“每一个 ServiceRegistration 对象有一个关联的 ServiceReference 对象”。这个关联没有一个硬关系，但可以从某种方式推断出来。

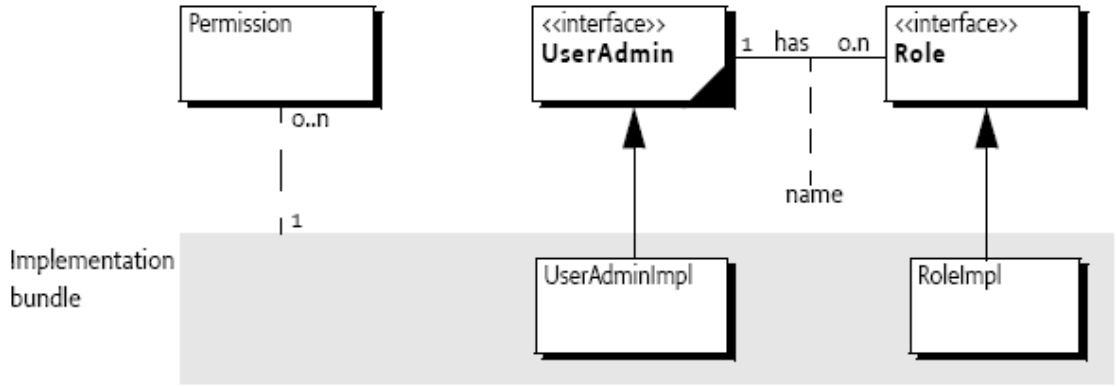
当一个关系使用一个名称或对象标识，它使用点垂线并使用一个框或描述连到这个线上。下图显示了使用一个 name 标识一个 UserAdmin 类和 Role 类的关系。这样的关系通常使用字典或 Map 对象来实现。

Associations symbol



Bundles 是一些在普通应用系统编程可见的实体。比如，当一个 Bundle 停止时，它的所有服务将被卸载。因此，类/接口成对的显示在矩形中。

Bundles



1.3.5 关键词

这个规范需要以一致的方式使用 may，should 和 must。他们意思描述如下。

●must——一个绝对的需求。框架实现和 Bundle 都有义务来完成遵循规范的功能。

●should——推荐。强烈建议遵循这个描述，但也可以不遵循。

●may 或 can——可选的。

1.4 版本信息

这个文档指定了 OSGi 服务平台的核心规范的第四版本。它将后向兼容 1，2 和 3 版本。

所有的安全、模块、生命周期和服务层是 Framework 规范的一部分。

在这个规范中，组件有他们的规范版本，与文档发行版本独立。下表总结了这些议题包和规范版本的区别。当一个组件表示一个 Bundle 时，在 Import-Package 或 Export-Package 清单必须指定版本。

项	所在包	版本
Framework 规范	org.osgi.framework	V1.4
条件权限管理	org.osgi.service.condpermissionadmin	V1.0
包管理服务规范	org.osgi.service.packageadmin	V1.2
权限管理服务规范	org.osgi.service.permissionadmin	V1.2
启动级别服务规范	org.osgi.service.startlevel	V1.1
URL 处理服务规范	org.osgi.service.url	V1.0

1.5 参考文档

[1] Bradner, S., Key words for use in RFCs to Indicate Requirement Levels
http://www.ietf.org/rfc/rfc2119.txt, March 1997.
[2] OSGi Service Gateway Specification 1.0, May 2000
http://www.osgi.org/resources/spec_download.asp
[3] OSGi Service Platform, Release 2, October 2001
http://www.osgi.org/resources/spec_download.asp
[4] OSGi Service Platform, Release 3, March 2003
http://www.osgi.org/resources/spec_download.asp
[5] Lexical Structure Java Language
http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html

2 安全层（V1.4）

2.1 介绍

OSGi 安全层是 OSGi 服务平台一个可选的层次。它基于 Java2 安全架构。它提供了部署和管理应用系统的基础设施，这些应用系统必须由执行环境严格控制。

2.1.1 要点

●细粒度——在 OSGi 运行的应用系统控制不需允许对这些应用提供细粒度控制。

●可管理——安全层本身没有定义控制应用系统的 API。安全层的管理留给了生命周期层。

●可选的——安全层是可选的。

2.2 安全概述

框架安全层模型基于 Java2 规范。如果执行了安全检测，他们必须根据 Java2 安全架构来操作。该文假设用户熟悉了 Java2 安全规范。安全层是可选的。

2.2.1 代码认证

OSGi 服务平台可以通过以下方式来认证代码：位置和签名。在更高的层次，它们被定义为服务，可以管理认证的代码单元的授权。这些服务是：

●权限管理服务——基于全路径字符串的权限管理。

●条件权限管理服务——使用条件模型管理权限，这些条件可以检测位置和签名。

对于签名，需要的 Jar 文件都必须经过签名。

2.2.2 可选安全

框架运行的 Java 平台必须为 Java2 权限提供 Java 安全 API。在资源受限的平台，这些 Java 安全 API 可能仅是空实现，以允许 Bundle 类被加载并执行，但这些 API 不会执行真正的安全检测。这些空实现必须是：

●checkPermission——不抛出 SecurityException 返回。

●checkGuard——不抛出 SecurityException 返回。

●implies——返回 true。

这个行为允许 Bundle 代码拥有所有权限被运行。

2.3 数字签名的 Jar 文件（略）

2.4 参考文档

[6] RFC 2253
http://www.ietf.org/rfc/rfc2253.txt
[7] X.509 Certificates
http://www.ietf.org/rfc/rfc2459.txt
[8] Java 2 Security Architecture
Version 1.2, Sun Microsystems, March 2002
[9] The Java 2 Package Versioning Specification
http://java.sun.com/j2se/1.4/docs/guide/versioning/index.html
[10] Manifest Format
http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR%20Manifest
[11] Secure Hash Algorithm 1
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf
[12] RFC 1321 The MD5 Message-Digest Algorithm
http://www.ietf.org/rfc/rfc1321.txt
[13] RFC 1421 Privacy Enhancement for Internet Electronic Mail
http://www.ietf.org/rfc/rfc1421.txt
[14] DSA
http://www.itl.nist.gov/fipspubs/fip186.htm
[15] RSA
http://www.ietf.org/rfc/rfc2313.txt which is superseded by
http://www.ietf.org/rfc/rfc2437.txt
[16] Public Key Cryptography Standard #7
http://www.rsasecurity.com/rsalabs/node.asp?id=2129
[17] Unicode Normalization UAX # 15
http://www.unicode.org/reports/tr15/
[18] Understanding and Deploying LDAP Directory Services
ISBN 1-57870-070-1

3 模块层（V1.4）

3.1 介绍

标准 Java 平台只提供打包、部署和基于 Java 应用系统和组件的验证。因此，很多基于 Java 的项目，比如 JBoss 和 NetBeans，都使用自定义的类装载器打包、部署和验证应用系统与组件，来创建自定义的面向模块的层。OSGi 框架为 Java 模块化提供了一个通用的和标准的解决方案。

3.2 Bundle

Framework 定义了模块化的单元，称为 Bundle。一个 Bundle 由 Java 类和其它资源组成，他们共同为终端用户提供功能。Bundle 能够以友好的的方式定义 Bundle 引用与导出来共享 Java 包。

在 OSGi 服务平台，Bundle 是基于 Java 应用系统部署的唯一实体。

一个 Bundle 作为一个 JAR 文件部署。JAR 文件以标准的基于 ZIP 的文件格式来存储应用系统和它们的资源。这种格式由 ZIP 文件格式定义。

一个 Bundle 是一个 JAR 文件，它：

●包含提供某些功能的必须的资源。这些资源可能是 Java 编程语言的类文件，也可能是其它数据如 HTML 文件、帮助文件和图标等。一个 Bundle 的 JAR 文件也可以嵌入额外的 JAR 文件，额外的 JAR 文件可作为类或资源。然而，这并不是递归的。

●包含一个清单（Manifest）文件，它描述了 JAR 文件的内容和提供了关于 Bundle 的信息。这个文件使用标题来指定一些信息，框架需要这些信息来正确的安装和激活一个 Bundle。比如，它描述了对其它资源的依赖，比如 Java 包，这些必须在 Bundle 运行之前可用。

●可以在 JAR 文件或一个子目录的 OSGI-OPT 文件夹包含可选文档。文件夹中的任何信息都是可选的。比如，OSGI-OPT 目录用于存储 Bundle 的源代码。管理系统可能会在 OSGi 服务平台中删除这些信息以节省空间。

一旦一个 Bundle 启动了，它便提供功能并将服务暴露给其它装载在 OSGi 服务平台的 Bundle。

3.2.1 Bundle 清单头部

一个 Bundle 可以在清单文件中包含它本身的描述信息，这个清单文件在它的 JAR 文件的 META-INF/MANIFEST.MF 中。

Framework 定义了 OSGi 清单的头部，如 Export-Package 和 Bundle-Classpath，Bundle 开发人员使用它们来提供一个 Bundle 的描述信息。清单头部必须严格遵循 Manifest 格式定义的清单头部规则。

一个 Framework 实现必须：

●处理清单的主要节。清单的不同节仅在 Bundle 签名验证是使用。

●忽略不认识的清单头部。Bundle 开发人员必须定义所需的额外清单头部。

●忽略不认识的特性和指令。

所有指定的清单头部在以下小节中列出。所有头部都是可选的，除非特别指定。

3.2.1.1 Bundle-ActivationPolicy: Lazy

Bundle 激活策略指定了 Framework 在启动 Bundle 时如何激活 Bundle。

3.2.1.2 Bundle-Activator: com.acme.fw.Activator

Bundle 激活器头部定义了用于启动和停止 Bundle 的类。

3.2.1.3 Bundle-Category: osgi,test,nursery

Bundle 种类头信息存放用逗号分开的种类名称列表。

3.2.1.4 Bundle-Classpath: /jar/http.jar

Bundle 类路径头信息定义了用逗号隔开的包含类和资源 JAR 文件路径或目录列表。而'.'句号指定了 Bundle 的 JAR 文件的根目录。句号也是默认的。

3.2.1.5 Bundle-ContactAddress: 2400 Oswego Road Austin,TX74563

Bundle 联系地址提供了 Bundle 提供商的联系地址。

3.2.1.6 Bundle-Copyright: OSGi(c) 2002

Bundle 版权头信息指定了 Bundle 的版权信息。

3.2.1.7 Bundle-Description: Network Firewall

Bundle 描述头信息定义了这个 Bundle 的简短描述。

3.2.1.8 Bundle-DocURL: http://www.acme.com/Firewall/doc

Bundle 的 DocURL 头信息定义了关于该 Bundle 的文档的 URL。

3.2.1.9 Bundle-Localization: OSGI-INF/l10n/bundle

Bundle 本地化头信息包含了 Bundle 可以找到的本地化文件所在的位置。默认的值是 OSGI-INF/I10n/bundle。默认的翻译是 OSGI-INF/I10n/bundle/bundle_de. Properties，OSGI-INF/I10n/bundle_nl.properties 等。

3.2.1.10 Bundle-ManifestVersion: 2

Bundle 清单版本头信息定义了 Bundle 遵循的规范定义的规则。Bundle 清单版本头信息决定 Bundle 是否要遵循这个规范的规则。对于 R3 的 Bundle，默认是 1 而对于 R4 以后版本则为 2。OSGi 服务平台在以后的版本可以在这个头信息定义一个大点的数字。

3.2.1.11 Bundle-Name: Firewall

Bundle 名称头信息定义了这个 Bundle 的可读的名称。它必须是一个简短的、容易理解的可包含空格名称。

3.2.1.12 Bundle-NativeCode: /lib/http.DLL; osname=QNX; osversion=3.1

Bundle 本地代码头信息包含了这个 Bundle 包含的本地代码库的信息。

3.2.1.13 Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0

Bundle 要求的执行环境包含一个逗号分开的执行环境列表。

3.2.1.14 Bundle-SymbolicName: com.acme.daffy

Bundle 的特征名称头信息为该 Bundle 指定了一个独一无二的、不可本地化的名字。这个名字必须基于反向域名规则。这个头信息是必须设置的。

3.2.1.15 Bundle-UpdateLocation: http://acme.com/bundle.jar

Bundle 更新位置头信息指定一个 URL，更新的 Bundle 可以从该 URL 下载。如果 Bundle 是可以更新的，这个位置必须使用，如果存在，将获取更新的 JAR 文件。

3.2.1.16 Bundle-Vendor: OSGi Alliance

Bundle 卖主头信息包含了 Bundle 卖主的易理解的描述。

3.2.1.17 Bundle-Version: 1.1

Bundle 版本信息指定了这个 Bundle 的版本。默认值是 0.0.0。

3.2.1.18 DynamicImport-Package: com.acme.plugin.*

动态引用包头信息包含了一个逗号分开的包名称列表，这些包在需要时会动态的加载。

3.2.1.19 Export-Package: org.osgi.util.tracker; version=1.3

导出包的头信息包含导出的包的列表。

3.2.1.20 Export-Service: org.osgi.service.log.LogService

已经作废。

3.2.1.21 Fragment-Host: org.eclipse.swt; bundle-version="[3.0.0, 4.0.0)"

宿主片段头信息定义了该片段的宿主 Bundle。

3.2.1.22 Import-Package: org.osgi.util.*;version=1.4

引用包头信息定义了该 Bundle 引用的包。

3.2.1.23 Import-Service: org.osgi.service.log.LogService

已作废。

3.2.1.24 Require-Bundle: com.acme.chess

依赖的 Bundle 头信息指定了该 Bundle 依赖的 Bundle。

3.2.2 头信息值语法

每一个清单头信息都有自定义的语法。在所有情况，这个语法在 W3C EBNF 中定义。以下节将定义一个通用的字符。

3.2.3 通用的头信息语法

很多清单头信息值都使用相同语法。这个语法由以下成分组成：

```
header          ::= clause(',' clause)*
clause          ::= path(';' path) *
                        (';' parameter)*        //查看 V1.3.2 版本
```

一个参数（Parameter）可以是一个指令（Directive）或一个特性（Attribute）。一个指令是 Framework 包含的语义。一个特性用于匹配和比较。

3.2.4 版本

版本规范用于不同的场合。一个版本字符串具有以下语法：

version ::=

major('.' Minor('.' Micro ('.' qualifier)?)?)?

major ::= number

minor ::= number

micro ::= number

qualifier::= (alphanu | '_' | '-')＋

一个版本字符不能包含任何空白。默认是 0.0.0。

3.2.5 版本范围

一个版本范围是基于数据闭包概念而描述的版本范围。其语法如下：

version-rang ::= interval | atleast

interval ::= ('[' | '(') floor ',' ceiling (')' | ']')

atleast ::= version

floor ::= version

ceiling ::= version

如果一个版本范围指定一个单一版本，它必须作为[version, ∞)范围来解释。对于一个没有指定的版本范围，默认为 0，即[0.0.0, ∞)。

需要注意的是在版本范围中逗号必须用双引号修饰。比如：

Import-Package: com.acme.foo; version=" [1.23, 2)",

com.acme.bar; version=" [4.0, 5. 0)"

3.2.6 过滤语法

OSGi 广泛的使用过滤表达式。过滤表达式允许你来简单的描述一个约束。

一个过滤字符串的语法是基于 LDAP 搜索过滤的字符串表达式（在[23]LDAP 搜索过滤字符串表达式）。在 RFC2254 需要注意的是，LDAP 搜索过滤的字符串表达式替换了 RFC1960，不过它仅添加了扩展的匹配功能，且目前 OSGi Framework 的 API 没有支持它。

LDAP 搜索过滤字符串表达式使用一个前缀格式并用以下语法定义：

filter ::= '(' filter-comp ')'

filter-comp ::= and | or | not | operation

and ::= '&' filter-list

or ::= '|' filter-list

not ::= ‘!’ filter
filter-list ::= filter
operation ::= simple | present | substring
simple ::= attr filter-type value
filter-type ::= equal | approx | greater | less
equal ::= ‘=’
approx ::= ‘~=’
greater ::= ‘>=’
less ::= ‘<=’
present ::= attr ‘=*’
substring ::= attr ‘=’ initial any final
initial ::= () | value
any ::= ‘*’ start-value
start-value ::= () | value ‘*’ start-value
final ::= () | value
value ::= <see text>

‘attr’是一个字符串表示一个特性，或者是属性（Properties）的 Key。特性名称大小写不敏感，也就是说，cn 和 CN 都是指向相同的特性。‘attr’不能包含等号、大于号、小于号、‘～’，左右括号。它可以包含空格但头尾的空格必须忽略。

‘value’是一个值的字符串表达式，或者是另一个值的一部分，将用来与过滤表达式的值进行比较。

如果值包含了‘\’，‘*’，‘（’或‘）’一个这样的字符，则这些字符必须在前面使用反斜杠‘\’。空格意义重大。空格字符在 Character.isWhiteSpace 中定义。

虽然 Substring 和存在产生式可以引入‘attr=*’结构表达式，不过这种表达式仅用于演示一个存在性过滤。

Substring 产生式只能对 String 类型、String 集合和 String 数组的属性起作用。在其它情况下，其返回值总为 false。

约等于匹配过滤表达式是自定义的实现，但至少应该忽略大小写和空格的差异。

在过滤器指定的值用于与过滤器过滤的属性（Properties）的值进行比较。这些值的比较不是直接的。字符串比较与数字比较不同，且一个属性可能有多个值。属性的那些 Keys 总是一些字符串对象，因此大小写不敏感的特性（attr）可以用来获取属性值。

属性值的对象类型在比较类型中定义。属性值必须是包含在以下几种类型的：

type ::= scalar | primitive | collection | array
scalar ::= String | Integer | Long | Float

| Double | Byte | Short
| Character | Boolean

primitive ::= int | long | float | double | byte | short | char | Boolean

array ::= <Array of primitive> | <Array of scalar>

collection ::= Collection of scalar

比较将使用以下规则：

●String——使用字符串比较。

●Integer,Long,Float,Double,Byte,Short,Character 对象和 Primitives——使用数字比较。

●Boolean 对象——使用 Boolean.valueOf(v).booleanValue()比较。

●数组或集合元素——比较由每一个元素的类型决定。

数组和集合元素可能是 Scalar 类型的混合。数组和集合元素也可能是 null。

如果属性值的类型不是上述的类型，且类型有一个只有一个字符串参数构造器的类型，Framework 必须将属性值作为字符串参数构造一个对象来使用以下规则比较：

●Comparable 对象——通过 Comparable 接口比较。

●其它对象——使用“=”比较。

如果无法使用上述比较规则，比较的结果总为 false。

如果过滤器匹配到至少一个属性的值，则它将匹配该属性。

比如，Dictionary dict = new Hashtable();

dict.put(“cn”, new String[] {“a”, “b”, “c”});

则 dict 将匹配一个使用 cn=a 或 cn=b 的过滤器。

3.3 可执行环境

一个被限制在一个或多个可执行环境的 Bundle 必须配置一个头信息来指定执行环境依赖。这个头信息是 Bundle-RequiredExecutionEnvironment。这个头部信息的语法是用逗号分隔的执行环境列表。

Bundle-RequiredExecutionEnvironment ::=

ee-name (‘,’, ee-name)*

ee-name ::= <defined execution environment name>

比如：

Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0, OSGi/Minimum-1.1。

如果一个 Bundle 在清单包含这个头信息，则 Bundle 只能使用所有执行环境合适的子集的方法。Bundle 应该列出它运行的执行环境。

3.3.1 执行环境的命名规则

执行环境需要一个合适的名称，所以：

●一个 Bundle 可以要求一个 Framework 在安装之前提供某个执行环境。

●提供 Framework 提供的执行环境的信息。

执行环境名称由除了空格字符和逗号字符的任何字符组成。OSGi 联盟已经定义了一些执行环境名称。

将来命名架构将使用 J2ME 配置和 Profile 名称。现在没有对这些命名架构的清晰定义不过相似的命名已经在多个规范使用了。

J2ME 架构使用一个配置和一个 Profile 名称来指定一个执行环境。OSGi 联盟将组合这两个名称到一个单一的执行环境名称。

现在在 J2ME 已经存在一些可执行环境了，这些在服务平台服务器也可能是有用的。执行环境的值得头信息必须与这些规范兼容。

一个 J2ME 执行环境名称由配置名称和 Profile 名称组成。在 J2ME，有两个不同的系统属性，即 micoredition.configuration 和 microedition.profiles。

比如，Foundation 的 Profile 使用 CDC-1.0/Foundation-1.0 作为执行环境名称。这个结构遵循以下规则：

ee-name= [<configuration> '-' <version> '/'
<profile> '-' <version>]

配置和 Profile 名称由 JCP 或 OSGi 联盟定义。如果一个执行环境没有一个配置或 Profile，'profile'部分用于标识执行环境的名称。这个规则并不是标准化的。

Name	**Description**
CDC-1.0/Foundation-1.0	J2ME Foundation Profile
OSGi/Minimum-1.1	OSGi EE，OSGi 框架实现的最小集
JRE-1.1	Java 1.1.x
J2SE-1.2	Java 2 SE 1.2.x
J2SE-1.3	Java 2 SE 1.3.x
J2SE-1.4	Java 2 SE 1.4.x
J2SE-1.5	Java 2 SE 1.5.x
JavaSE-1.6	Java SE 1.6.x
PersonalJava-1.1	Personal Java 1.1
PersonalJava-1.2	Personal Java 1.2
CDC-1.0/PersonalBasis-1.0	J2ME Personal Basis Profile

CDC-1.0/PersonalJava-1.0	J2ME Personal Java Profile

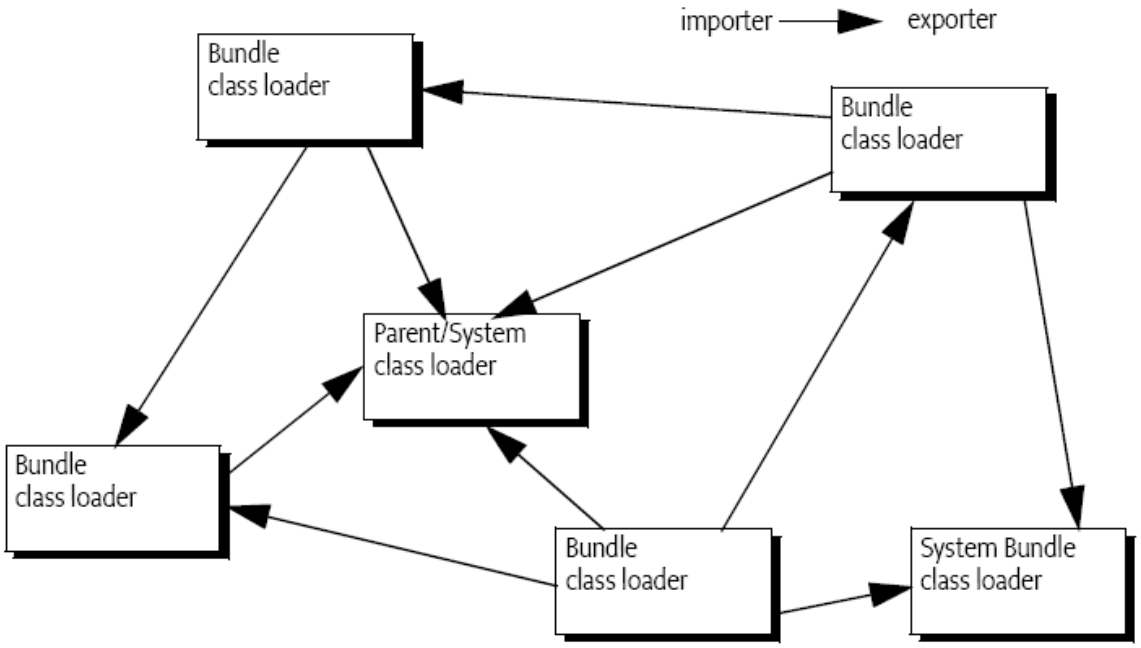
从 BundleContext.getProperty(String)获取的 org.osgi.framework.executionment 属性必须包含 Framework 实现的用逗号分开的执行环境列表。这个属性定义为可变的。一个 Framework 实现不应该缓存这些信息，因为 Bundle 可能会在任何时间改变系统属性。这个可变性的目的是在运行时测试和扩展执行环境。

3.4 类装载架构

很多 Bundle 可能共享一个虚拟机。在这个虚拟机里，Bundle 可以向其它 Bundle 隐藏或共享包和类。

隐藏和共享包的关键机制是 Java 类加载器，类加载器可以使用预先定义的规则从 Bundle 类空间（Bundle-space）的子集加载类。每一个 Bundle 都有一个单独的类加载器。那些类加载器和其它 Bundle 类加载器形成一个类加载代理网络。

Class Loader Delegation model



类加载器可以从以下位置加载类和资源：

●启动类路径——启动类路径包含了 java.*的包和它的实现包。

●Framework 类路径——Framework 通常会为框架实现和关键服务接口类提供一个单独的类加载器。

●Bundle 空间——Bundle 空间由 Bundle 本地的 JAR 文件和与 Bundle 紧密联系的 JAR 文件（如片段、片段 Bundle）。

一个类空间是一个给定的类加载器能够到达的所有类。因此，给定的 Bundle 的一个类空间包含的类来自：

●父类加载器，一般的 java.*包来自启动类路径。

●引用的包。

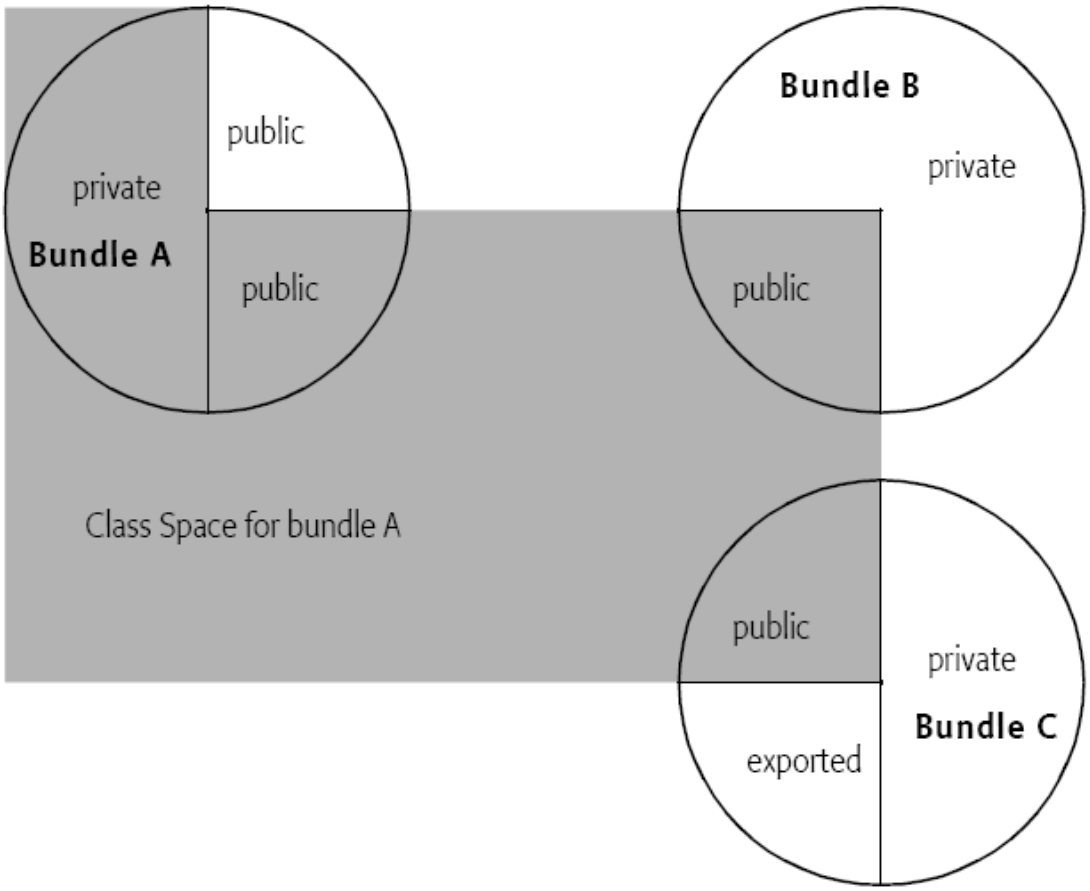
●依赖的 Bundle。

●Bundle 的类路径。

●附加的片段。

一个类空间必须是一致的，因此它从不会包含两个使用同一标识名称的类。然而，在一个 OSGi 平台不同的类空间可以包含使用同一标识名称的类。模块层允许同一类的多个版本被装载到同一虚拟机。

Class Space



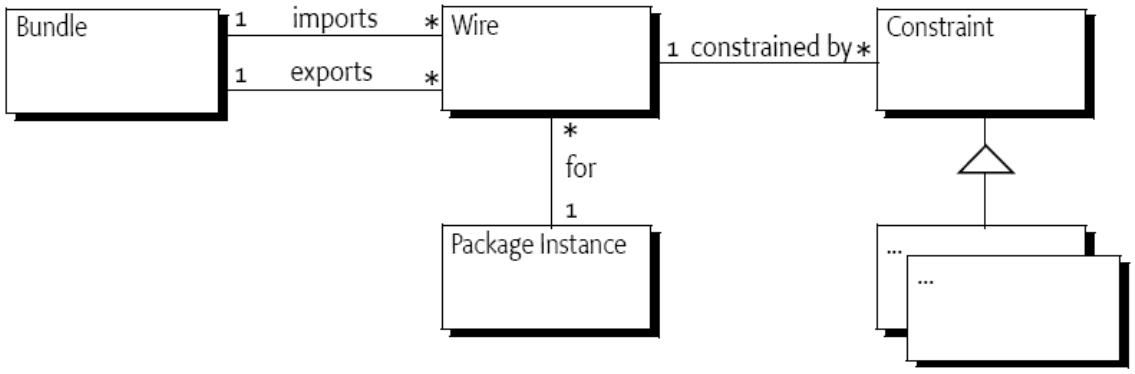
因此 Framework 为类型装载的一些职责。在使用一个 Bundle 之前，它必须解析 Bundle 要求的共享包。然后选择最好的方式来装配。

3.4.1 解析

Framework 必须解析 Bundle 依赖。解析是引用者连线到导出者的过程，即满足约束的过程。这个过程在装载和执行 Bundle 的代码之前发生。

一个连接线是导出者和引用者间实际的联系。一个连接线关联一些约束，它们由导出者和引入者清单头文件定义。一个有效的连接线是满足所有约束的连接线。下图描述了连线模型的类结构。

Example class structure of wiring



3.5 解析元素据

在以下小节定义了为解析器（Resolver）提供元素据的清单头信息。

3.5.1 Bundle-ManifertVersion

一个 Bundle 清单文件必须在 Bundle-ManifestVersion 头信息标明 OSGi 清单头信息语法的版本。Bundle 使用框架规范的这个版本必须指定这个头信息。头信息的语法为：Bundle-ManiferVersion ::= number（请看 V1.3.2）。

V1.3 框架 Bundle 清单版本必须是 2。用以前规范的清单语法编写的 Bundle 清单其清单版本必须是 1，虽然在清单中没有表示。因此，在这个头信息中任何大于 2 的版本是不可用的，除非 Framework 明确支持最近版本。

OSGi 框架实现必须支持没有 Bundle-ManifestVersion 头信息的 Bundle 清单并假设 Framework 1.2 也是支持的。

V2 Bundle 清单必须指定 Bundle 的特征名称。他们不需要指定 Bundle 的版本，因为这个版本有一个默认值。

3.5.2 Bundle-SymbolicName

Bundle 特征名称清单头信息是一个强制的头信息。Bundle 特征名称和 Bundle 版本允许来唯一标识 Framework 的一个 Bundle。也就是说，一个使用给定特征名称和版本的 Bundle 被看作与另一个有相同特征名称和精确版本的 Bundle 相等。

安装一个已经存在的特征名称和版本的 Bundle 必须导致失败。

一个 Bundle 从开发人员获取唯一的特征名称。它必须遵循以下语法：

Bundle-SymbolicName ::= symbolic-name (';' parameter) *

框架必须接受 Bundle 特征名称头信息中的如下指令：

●singleton——指定 Bundle 只能有一个版本运行。一个 true 值指定这个 Bundle 是一个单件的 Bundle。默认值是 false。当多个版本的单件的使用相同特征名称的 Bundle 被安装时，框架至多只能解析一个 Bundle。单件的 Bundle 不能影响其它使用相同特征名称不是单件的 Bundle。

●fragmented-attachment——定义一个片段如何被添加。在这个指令只允许一下值：

A) always——片段在宿主解析后或在解析过程时的任何时候添加。

B) never——不允许添加片段。

C) resolve-time——片段必须在解析过程中添加。

如：Bundle-SymbolicName: com.acme.foo; singleton := true。

3.5.3 Bundle-Version

Bundle 版本是一个可选的头信息，默认是 0.0.0。

Bundle-Version ::= version

如果 minor 或 micro 版本部分没有指定，它们默认为 0。如果 qualifier 部分没有指定，默认为空字符串。

版本是可以比较的。他们的比较实用数字和对 major、minor 和 micro 各个部分按顺序比较，最后使用字符串类的 compareTo 方法比较 qualifier。只有当这四个部分都相同时，才认为两个版本一样。

版本例子：Bundle-Version: 22.3.58.build-345678。

3.5.4 引用包头信息

引用包头信息定义了共享包引用约束。引用包头信息的语法如下：

Import-Package ::= import (‘,’ import)* （引用包的定义）

import ::= package-names (‘;’ parameter)* （引用的定义）

package-names ::= package-name (‘;’ package-name)*

头信息允许引用多个包。一个引用（import）定义是 Bundle 一个包的描述。这个语法允许使用分号隔开的多个包名，以简短的信使描述。

引用包指令（parameter ::= directive | attribute）是：

●resolution——如果它的值是 mandatory（默认），则指定所有的包都必须被解析。如果强制引用的包不能解析，则 Bundle 解析必须是失败。如果它的值是 optional，则引用包是可选的。

开发人员可以指定任意匹配特性。以下是预定义的匹配特性：

●version——一个版本范围来选择导出者的包版本。这个语法必须遵循版本范围。如果这个特性没有定义，则假设为[0.0.0, ∞)。

●specification-version——这个特性是 version 特性的别名，仅用于从前一个版本移植。如果 version 特性指定了，则这两个特性必须相同。

●bundle-symbolic-name——导出包的 Bundle 的特征名称。对于片段 Bundle，这必须是宿主 Bundle 的特征名称（这意味着，如果你引用的包是来自一个片段，那么你声明这个引用的时候必须指向它的宿主 Bundle 的特征名称）。

●bundle-version——选择导出 Bundle 的版本范围。默认值是[0.0.0, ∞)。如果是片段 Bundle，则这个版本必须是宿主 Bundle 的版本。

为了允许引入包，一个 Bundle 必须有 PackagePermission[<package-name>,IMPORT]权限。

在以下错误发生时将会终止一个 Bundle 的安装或更新：

●一个指令或特性出现了多次。

●同一个包有多个引用定义。

以下是一个正确定义的例子：

```
Import-Package: com.acme.foo; com.acme.bar;
        version=”[1.23, 1.24]”
        resolution := mandatory
```

3.5.5 导出的包

导出包头信息的语法与导入包头信息语法类似。只是指令和特性有所不同。

Export-Package ::= export (‘,’ export)*

export ::= package-names (‘;’ parameter)*

package-name ::= package-name (‘;’ package-name)

头信息允许导出多个包。一个导出定义是一个 Bundle 导出一个包的描述。这个语法通过使用分号将包名隔开来允许在一个子句声明多个包。当需要不同特性时，同一个包的多个导出定义也是允许的。

导出的指令有：

●uses——一个用逗号分隔的导出的包的包名。需要注意的是逗号的使用需要在双引号中。

●mandatory——一个用逗号分隔的特性名称。需要注意的是逗号的使用需要在双引号中。一个 Bundle 导入一个包时必须使用一个值指定 mandatory 特性，这个值将匹配来解析导出的包（一个导出可以声明导出相关的特性标识，如 myAttr=UIShell，那么若对这个导出指定了 mandatory=”UIShell,A,B,C”，则引用的定义必须也指定一个 myAttr=UIShell 后才能引用这个导出）。

●include——逗号分开的类名列表，这些类必须对引用者可见。

●exclude——逗号分开的类名列表，这些类对引用者不可见。

以下特性是这个规范的一部分：

●version——默认值是 0.0.0。

●specification-version——version 的别名，同导出定义。

此外，你可以指定任意匹配特性。

Framework 将使用以下特性来与导出的包自动关联：

●bundle-symbolic-name——导出的包的特征名称。对于片段 Bundle，这是宿主 Bundle 的特征名称（即一个片段的导出定义，其对应的 Bundle 特征名称必须是它的宿主 Bundle 特征名称）。

●bundle-erison——导出包的版本。对于片段 Bundle，这是宿主 Bundle 的版本。

当以下条件为 true 时，一个 Bundle 安装或更新必须被终止：

●一个指令或特性重复多次。

●在导出包头信息指定了 bundle-symbolic-name 或 bundle-version 特性。

一个导出定义并不意味着一个自动导入的定义。一个导出一个包但没有引用该包的 Bundle，将从其类路径获取那个包。这样的一个导出定义仅能被其它 Bundle 使用，且导出 Bundle 不能使用另一个 Bundle 的替代包。

为了导出一个包，一个 Bundle 必须具有 PackagePermission[<package>, EXPORT]权限。

示例：

Export-Package: com.acme.foo; com.acme.bar; version=1.23

3.5.6 导入和导出一个包

导出一个包并不意味着对那个包的引用。这种分离的原因是它使得一个 Bundle 可以不需要考虑导出包可能被解析器使用另一个 Bundle 的相同的包替换情况下，向另一个 Bundle 提供一个包（如果导出一个包意味着对那个包的引用，则这个引用可能在解析的时候使用另一个 Bundle 的导出）。一个应用系统由一堆紧密相连的 Bundle 组成，这些 Bundle 向其它 Bundle 提供实现的包，这是一种常见的情况。

包的替换对于 Bundle 的互操作性是至关重要的。在 Java，只有当 Bundle 为同样的类使用同样的类加载器时，Bundle 才可以互操作。因此，两个都导出相同包但没有引用它的 Bundle 不能从那个包共享对象。这对于一个像服务层的协调机制是非常重要的。如果服务的类和接口来自同一个类加载器时，Bundle 才能使用相同的服务对象。

Bundle 引用的导出包，应该允许解析器使用包含接口和其它共享类型进行替换。这种可替代性允许 Bundle 通过服务注册表和其他机制实现互操作。此外，导入必须尽可能减少限制从而允许解析器有最大的灵活性。

3.5.7 早期版本 Bundle 的解析

那些没有使用 Bundle-ManiferVersion>=2 标记 Bundle 必须根据 R3 来处理这个头信息。更特别的，Framework 必须将 R3 头信息到合适的 R4 头信息：

●Import-Package——一个引用定义必须改变 specification-version 特性到 version 特性。一个没有使用特定版本的引用定义不需要被替换，因为与 R3 一样默认为 0.0.0。

●Export-Package——一个导出定义必须将 specification-version 特性变更到 version 特性。导出定义必须添加 uses 指令。uses 指令必须包含给定 Bundle 所有导入和导出的包。此外，如果包没有一个导入定义，那么一个使用给定版本对这个包的引用必须被添加。

●Dynamic-Package——一个动态引用定义没有被改变。

一个混合了以前版本语法和 2.0 版本语法 Bundle 清单是错误的且必须限制使这个 Bundle 安装失败。

specification-version 特性在 2.0 Bundle 清单版本已经被废弃。

3.6 约束解析

OSGi 框架包解析器提供了匹配引用和导出的一些机制。以下小节将详细描述这些机制。

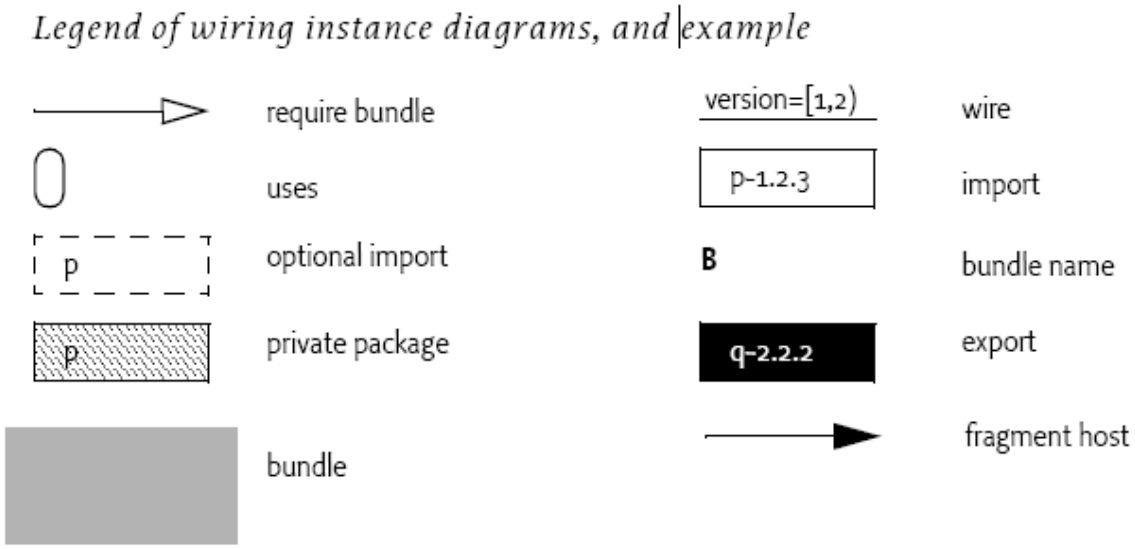
3.6.1 图和语法

图由若干连线的节点形成。连接线和节点带大意义重大的信息表示。在接下来的小节，以下约定将用于解释的更多的细节。

Bundle 被命名为 A,B,C…，即从 A 开始的大写字符。包被命名为 p,q,r,s,t…，换句话说，即从 p 开始的小写字符。如果一个版本是重要的，它将使用像这样的版本：q-1.0。语法 A.p 表示包定义，即 Bundle A 的包 p。

引用定义使用一个白盒表示。导出定义使用一个黑盒表示。那些没有导出也没有导入的包叫私有包，它们使用斜角线表示。

Bundle 集是一系列连线的盒子。约束表示在连接线上。



比如：

A: Import-Package: p; version="[1,2)"

Export-Package: q; version=2.2.2; uses := p

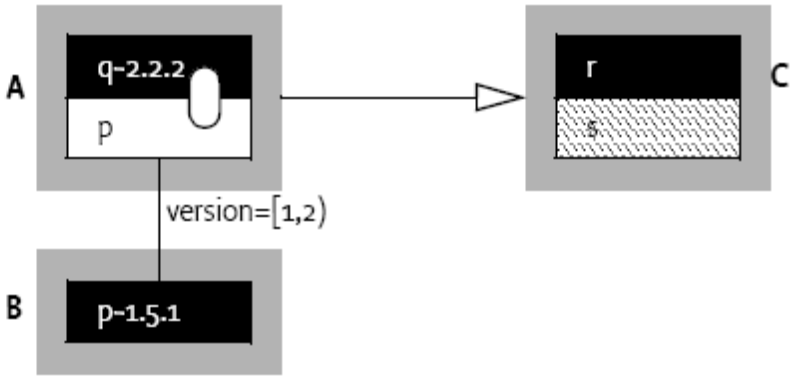
Require-Bundle: C

B: Export-Package: p; version=1.5.1

C: Export-Package: r

以下用图形方式表示相同的描述。

Example bundle diagram



3.6.2 版本匹配

版本约束是一个引用定义使用一个确切的版本或版本范围以此来匹配一个导出定义的机制。

版本范围用于兼容性。该规范没有定义兼容性政策，政策决定留给指定版本范围的引用者。一个版本范围即隐含着一个这样的兼容政策。

然而，大多数版本兼容策略是：

●major——一个不兼容的更新。

●minor——一个向后兼容的更新。

●micro——一个没有影响接口的更新，如修改 Bug。

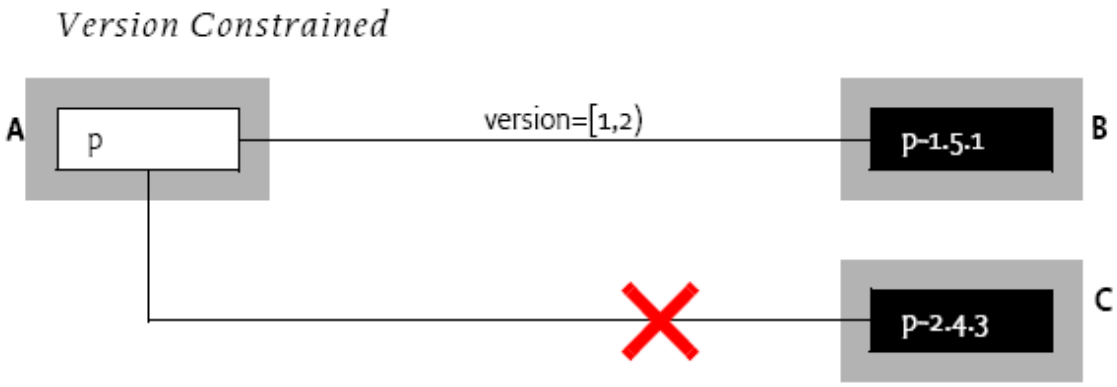
一个引用定义需要在 version 特性的值里面指定版本范围。

比如，以下引用和导出定义将能够正确解析，因为引用的版本范围与导出定义的版本匹配。

A: Import-Package: p; version="[1,2]"

B: Export-Package: p; version=1.5.1

下图显示一个约束如何排除导出者。



3.6.3 可选包

一个 Bundle 可以被指定为不要求正确解析一个包，但是如果该包可用的话，Bundle 可能会使用它。比如，日志是重要的，但缺少日志服务不应该阻止 Bundle 运行。

可选的引用可以通过两种方式指定：

●Dynamic Import——动态引用头信息用于在需要那个包时查找一个导出包。动态引用的关键使用场景是当一个 Bundle 不知道类名称时它需要用 Class.forName 来加载。

●Resolution 指令——这个在导入定义的指令可以指定为 optional。一个 Bundle 在合适的可选包不存在时，能够解析成功。

这两种机制的主要区别是在什么时候建立连接线。对于动态引用，在每次尝试装载那个包的类时，都尝试建立对其引用的连接；而对于 resolution 为 optional 时，仅在解析 Bundle 时建立连接。

以下示例也能够解析，即使 Bundle B 没有提供正确的版本。

A: Import-Package: p;

resolution := optional;

version = 1.6

B: Export-Package: p;

q;

version=1.5.0

一个使用了可选包 Bundle 的实现必须能够处理好引用包不可用的情况，也就是说，在这种情况下当引用了没有正确解析的包的类时将会引发异常。

3.6.4 包约束

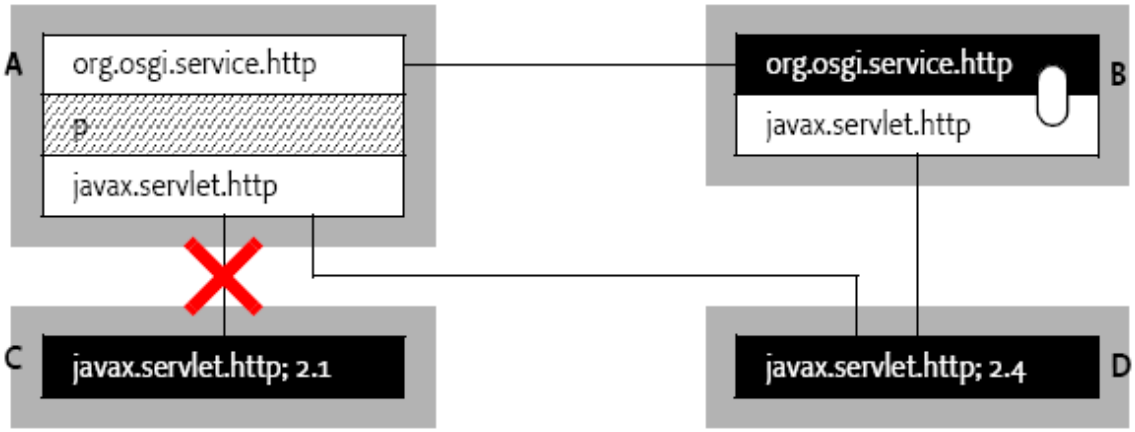
类可以依赖其它包的类。比如，当他们扩展了其它包的类，或这些其它包的类在方法签名中出现。它因此可以被称为一个包使用了其它包。这些包间依赖关系可以使用在导出包头信息的 uses 指令来建模。

比如，org.osgi.service.http 依赖 javax.servlet，因为它在 API 被使用了。在 org.osgi.service.http 因此在导出定义中必须包含 uses 指令，这个指令的值是 javax.servlet。

如果一个 Bundle 中每一个包只有一个导出，则类空间的一致性便能够得到保证。

比如，HttpService 实现需要扩展 javax.servlet.http.HttpServlet 基类的 Servlet。如果 HttpService Bundle 想要导入 2.4 版本且客户端 Bundle 需要导入 2.1 版本，一个类转换将会发生。

Uses directive in B, forces A to use javax.servlet from D

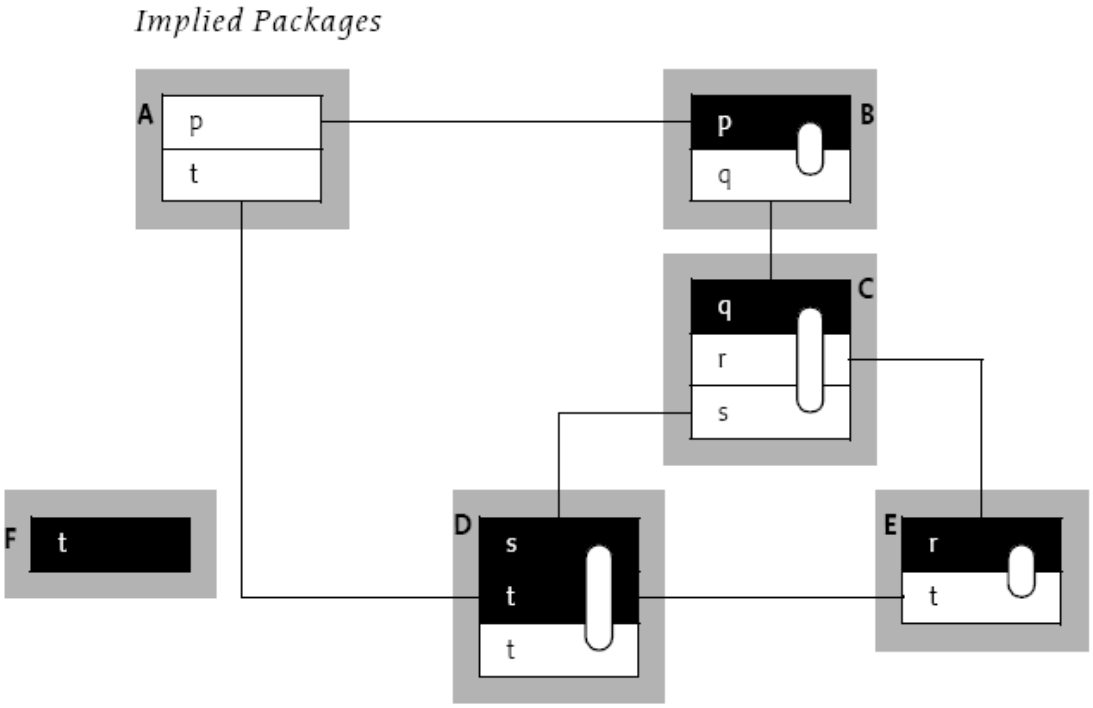


如果一个 Bundle 引用了导出 Bundle 的一个包，那么那个包的导出定义隐含着通过 uses 指令对其它包的约束。uses 指令列出了导出 Bundle 依赖的包，并以此强制解析器为引用也指定了这些依赖。这些约束确保了一堆 Bundle 为相同的包共享相同的类装载器。

当一个引用 Bundle 使用隐含的约束引用一个包时，解析器必须使用隐含的约束连接对导出 Bundle 的引用。这个导出的 Bundle 可能反过来隐含额外的约束等。一个单一引用的包到导出者的连线将因此隐含一堆的约束。名词“隐含的包约束”指的是递归的建立连接的完全的约束。隐含包约束不是自动引用的，相反，隐含包约束只能约束一个引用定义如何被解析。

比如，在下图，Bundle A 引用包 p。假设这个引用定义关联了 Bundle B。由于使用 uses 指令，这意味着包 p 有一个约束。此外，假设对包 q 的引用关联到 C，那么这意味着一个

约束将添加到包 r 和 s。照此类推，假设 C.s 和 C.r 分别的关联的 D 和 E。这些 Bundle 都为 Bundle A 添加隐含包。



图说明：A 引用 B 中的 p， B 依赖了 C 中的 q， C 依赖了 D 中的 s 和 E 中的 r，E 依赖了 D 中的 t。B 导出 p，C 导出 q，D 导出 s 和 t，E 导出 r，他们都使用了 uses 指令。A 引用 B 时，将隐式的引用了 B.p 在 uses 中声明的包，如果 B.p 的 uses 使用了 C，C 使用了 D 和 E，那么 A 将隐式的引用这些包。

为了维护类空间的一致性，Framework 必须确保 Bundle 的引用不会与 Bundle 隐含包发生冲突。

比如，这就意味着 Framework 必须确保引用 A.t 连接到 D.t。如果连接到 F.t 将破坏类空间的一致性。这个破坏发生是因为 Bundle A 使用 D 和 F 的类加载器都能够获得相同类名的对象。这将潜在的发生 ClassCastException 异常。或者，如果所有 Bundle 都连接到 F.t，那么这个问题也就不存在了。

另一种场景是在上上个图。Bundle A 引用了 Bundle B 的 HttpService 类。Bundle B 组合了 org.osgi.service.http 和 javax.servlet 且 A 约束的连线到 javax.servlet，这与 Bundle B 导出相同。

以下实例是 uses 指令能够正确解析的情况：

A: Import-Package: q; version=”[1.0,1.0]”

Export-Package p; uses=”q,r”

B: Export-Package: q; version=1.0

C: Export-Package: q; version=2.0

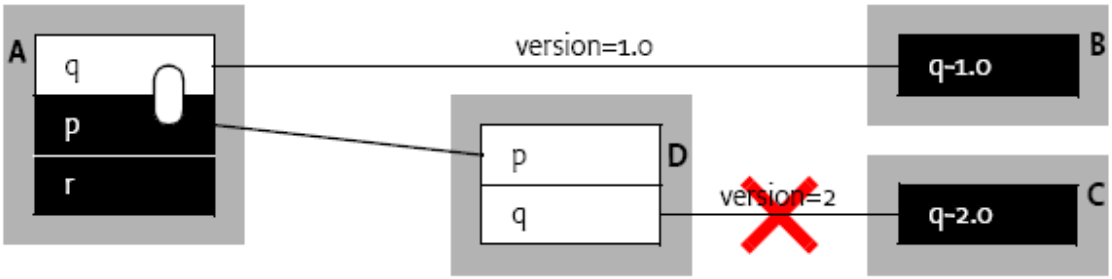
这些指令约束能够被正确解析是因为使用了版本约束使 A.q 可以连接到 B.q 而不是 C.q。

添加一个以下 Bundle D 将是不可能解析的：

D: Import-Package: p,q; version=2.0

包 D.p 必须连接到 A.p，因为 Bundle A 是唯一的导出者。然而，这隐含着包 q 的使用，由于它在 A.q 使用了 uses 指令。包 A.q 连接到 B.q-1.0。然而引入包 D.q 需要 2.0 版本，因此在不破坏类空间一致性情况下是不能正确解析的。

Uses directive and resolving



3.6.5 特性匹配

特性匹配是一个常用的机制，允许引用 Bundle 和导出 Bundle 来以声明的方式影响匹配过程。为了一个引用定义能够被一个导出定义解析，在引用定义特性的值必须与导出定义指定的特性的值匹配。默认的，如果导出定义包含了没有在引用定义出现的特性，这个比配不会被阻止的。导出定义的 mandatory 指令相关过来，它列出了 Framework 在导入定义必须匹配的特性。在 DynamicImport-Package 中定义的任何信息在解析时都会被忽略。

比如，下述语句将会被匹配。

A: Import-Package: com.acme.foo; company=ACME

B: Export-Package: com.acme.foo;

company=”ACME”;

security=false

特性值是可比较的字符串，除了 version 特性和 bundle-version 特性，他们使用版本范围比较。头部和尾部的空白将被忽略。

3.6.6 Mandatory 特性

有两个类型的特性：mandatory 和 optional。Mandatory 特性指定了在引入定义必须匹配。Optional 特性在他们没有被引用 Bundle 引用时是被忽略的。特性默认的是 Optional。

导出 Bundle 可以指定在导出定义中使用 mandatory 指令指定 Mandatory 特性。这个指令包含了逗号分开的特性名，它们必须在引用中指定来匹配。

比如，以下引用定义不能匹配导出定义，因为 mandatory 特性：

A: Import-Package: com.acme.foo; company=ACME

B: Export-Package: com.acme.foo;

company=”ACME”;

security=false;

mandatory := security

3.6.7 类型过滤

一个导出 Bundle 可以在导出定义使用 include 和 exclude 指令限制包中类是否可见。这些指令都是使用逗号分隔的类名列表。

类名不能包含包名且不能以.class 结束。也就是说，类 com.acme.foo.Daffy 就是列表的 Daffy。类名可以包含乘号“*”。

默认的对于 include 和 exclude 指令，他们的值是“*”，因此没有类和资源被排外，一个空列表不匹配任何的名字。如果 include 或 exclude 指令被指定，则默认值将被覆盖。

一个类只有在以下情况才是可见的：

●匹配 include 列表的一个项目。

●且不在 exclude 列表中。

在其它情况，装载或查找会引起失败，且一个 ClassNotFoundException 异常会被抛出。include 和 exclude 的顺序没有限制。

以下示例演示了一个 export 语句，且列表显示了他们的可见性。

Export-Package: com.acme.foo; include := “Qux*, BarImpl”;

Exclude := QuxImpl

com/acme/foo

QuxFoo visible

QuxBar visible

QuxImpl excluded

BarImpl visible

使用过滤时必须小心。比如，一个新版本的模块与一个早期的版本向后兼容，则它不能过滤那些没有被早期版本过滤的类和资源。此外，当模块化已有代码时，从引用包过滤类或资源可能会破坏包的使用。

以下例子中，使用标准格式定义的包需要在一个对指定类具有包访问权限的标准包中实现。

```
package org.acme.open;
public class Specified {
        static Specified implementation;
        public void foo() { implementation.foo(); }
}

package org.acme.open;
public class Implementation {
        public void initialize(Specified implementation) {
                Specified.implementation = implementation;
        }
}
```

实现类必须不能对外部 Bundle 可见，因为它允许这个实现是可以被指定的。通过排除 Implementation 类，只有导出的 Bundle 可以看到这个类。头信息的导出定义可以是：

Export-Package: org.acme.open; exclude := Implementation

3.6.8 供应商选择

供应商选择允许引用 Bundle 来选择哪个 Bundle 可以考虑为导出的 Bundle。供应商选择在引用者和导出者间没有指定约束时使用。引用者与它的导出者紧密耦合，典型的是测试的 Bundle。为了使连线不太脆弱，引用者可以选择的指定匹配的 Bundle 版本范围。

一个引用者可以使用 import 特性 bundle-symbolic-name 和 bundle-version 来选择一个导出者。框架自动为每一个导出定义自动提供这些特性。这些特性在导出定义中不应该指定。

导出定义 bundle-symbolic-name 包含在没有任何参数 Bundle-SymbolicName 头信息定义的 Bundle 特征名称。导出定义的 bundle-version 特性设置为头信息 Bundle-Version 的值，如果没有指定的话，默认为 0.0.0。

bundle-symbolic-name 作为一个特性匹配。bundle-version 作为一个版本范围匹配。引用的定义必须是一个版本范围而导出定义是一个版本。

A: Bundle-SymbolicName: A

Import-Package: com.acme.foo;

bundle-symbolic-name=B;

bundle-version=”[1.4.1, 2.0.0]”

B: Bundle-SymbolicName: B

Bundle-version: 1.41

Export-Package: com.acme.foo

以下语句不能匹配，因为 B 没有指定版本，其默认为 0.0.0。

A: Bundle-SymbolicName: A

Import-Package: com.acme.foo;

bundle-symbolic-name=B;

bundle-version=”[1.4.1, 2.0.0]”

B: Bundle-SymbolicName: B

Export-Package: com.acme.foo; version=1.4.1

通过特征名称选择导出 Bundle 会产生脆弱性，因为这将与 Bundle 的包产生硬耦合。比如，如果导出者最后需要重构一系列 Bundle，那么所有的导入都需要改变。其它任意匹配特性没有这个缺点，因为他们可以单独指定导出的 Bundle。

在 Bundle 重构引发 Bundle 特征名称的脆弱问题可以通过编写一个使用原始 Bundle 名称的辅助 Bundle 来部分的克服。

3.7 解析过程

解析是一个创建 Bundle 连接线的过程。连接线的约束通静态的定义在：

●引入和导出的包（DynamicImport-Package 头信息在这个阶段被忽略）。

●需要的 Bundle，将引用 Bundle 所有导出的包。

●片段，提供他们的内容和宿主定义。

在一个 Bundle 解析前，必须附上所有的片段。解析的过程是一个约束解析的算法，可以被描述为连接关系的需求。解析过程是一个迭代的过程，它将通过解析方案空间搜索。

如果一个模块具有对同一个包的引用和导出定义，框架将决定选择哪一个。

框架将首先尝试解析引用定义。可能会是以下结果：

●外部——如果这个包在另一个 Bundle 的导出定义中，那么将丢弃这个包的定义。

●内部——如果在这个模块的一个导出定义中，则这个模块的引用定义将被丢弃。

●没有解析——没有匹配的导出定义。

如果一个 Bundle 满足以下条件，它能够被解析：

●所有强制的引用都已经连接了。

●所有强制需要的 Bundle 都可用且他们的导出都连接了。

一个连线仅在以下情况满足下才创建：

●导入 Bundle 的版本范围与导出 Bundle 版本匹配。

●导入 Bundle 指定了导出 Bundle 所有强制的特性。

●所有导入 Bundle 的特性都与导出 Bundle 相应特性匹配。

●引用包到相同包的引用被连线到相同的导出者。

●连线到一个有效的导出 Bundle。

如果有多个选择，以下按权限从大到小列出偏好选择：

●一个解析的导出 Bundle 优于一个没有解析的。

●一个有高版本的导出 Bundle 优于一个低版本的。

●一个使用低 Bundle ID 的导出 Bundle 优于一个高的。

3.8 运行时类型装载

每一个在 Framework 的 Bundle 在没有解析前都不能有一个关联的类装载器。在 Bundle 解析后，Framework 必须为每一个不是片段 Bundle 的 Bundle 创建一个类装载器。Framework 也可能会延迟创建类装载器，直至真的需要。

每一个 Bundle 一个类装载器允许一个 Bundle 的所有资源具有对在其他 Bundle 的相同包的资源的包级访问权限。为了避免命名冲突，类装载器使用自己的命名空间提供个每一个 Bundle，且允许与其他 Bundle 共享资源。

类加载器必须使用在解析过程计算的连接线来查找合适的导出者。如果在引用中有一个类不能找到，清单的额外信息可以控制在其它地方对类和资源进行搜索。

以下节定义了影响运行时类装载的因素并定义了 Framework 装载一个类或资源遵循的准确的搜索顺序。

3.8.1 Bundle 类路径

Bundle 内部类路径依赖在 Bundle-Classpath 清单头信息声明。这个声明允许一个 Bundle 来使用一个或多个 JAR 文件或包含 bundle 的 JAR 文件的目录来声明内嵌的类路径。

Bundle-Classpath 清单头信息是用逗号隔开的文件名列表。一个文件名可以是：

●点（'.' \u002E），表示 Bundle 的 JAR 文件本身，默认值。

●JAR 文件的路径。

●目录。

Bundle-Classpath 清单头信息必须遵循以下语法：

Bundle-Classpath ::= entry (‘,’ entry)*

entry ::= target (‘;’ target)*

(‘;’ parameter)*

target ::= path | ‘.’

Framework 必须忽略不认识的参数。

如果一个 target 在需要时不能被找到，那么 Framework 必须忽略在 Bundle-Classpath 定义的这个 target，这可能会在 Bundle 解析后的任何时刻发生。然而，在这种情况，Framework 必须发布一个使用 INFO 类型的事件，并为每一个不能定位的项设置合适的消息。

当定位 Bundle 类路径的一个项时，Framework 必须尝试定位到与 Bundle 的 JAR 文件根目录相关的类路径。如果一个路径项不能定位，则 Framework 必须尝试定位到片段 Bundle 的类路径。附加的片段 Bundle 按 Bundle ID 升序搜索。这允许一个片段来提供项目，它们将被插入到宿主的 Bundle-Classpath。

以下是演示示例：

A: Bundle-SymbolicName: A

Bundle-Classpath: required.jar, optional.jar, default.jar

content…

required.jar

default.jar

B: Bundle-SymbolicName: B

Bundle-Classpath: fragment.jar

Fragment-Host: A

content…

required.jar

default.jar

在这个例子，Bundle A 有一个三个项的 Bundle-Classpath 头信息。required.jar 类路径项可能包含必须存在来向 Bundle 提供功能的类和资源。optional.jar 类路径项可能包含如果它存在的话，Bundle 将会使用的类和子资源。

default.jar 类路径项可能包含如果 optional.jar 不存在的话而使用的类和资源，但是 defualt.jar 的类和资源可以被 optional.jar 的类和资源覆盖。Bundle A 仅将 required.jar 和 default.jar 包和它一起打包。这允许片段 Bundle B 安装来向 Bundle A 提供 optional.jar 包。

片段 Bundle B 有一个使用一个项目的 Bundle-Classpath（fragment.jar）。当 Bundle A 解析后且片段 B 被附加到 A，那么 Bundle A 的类路径是：

required.jar， optional.jar， default.jar， fragment.jar

3.8.2 动态引用包

动态引用将在执行类型加载时匹配对应的导出定义，并且不影响模块解析。动态引用只能使用在那些没有建立连接线且不能以任何方式查找到定义的包。动态引用作为最后的手段使用。

Dynamic-Package ::= dynamic-description
(‘,’ dynamic-description)*
dynamic-description ::= wildcard-names (‘;’ parameter) *
wildcard-names ::= wildcard-name (‘;’ wildcard-name) *
wildcard-name ::= package-name
| (package-name ‘.*’)
| ‘*’

框架对动态引用没有使用任何指令。不过可以指定任意匹配特性。以下匹配特性由 Framework 提供：

●version——一个选择导出定义的版本范围。默认是 0.0.0。

●bundle-symbolic-name——导出 Bundle 的特征名称。

●bundle-version——导出 Bundle 的版本范围。

包可以被显示的命名，或者使用如 org.foo.*和*等的通配符表达式。通配符可以使用任何后缀，包括多个子包。

动态引用必须以指定的顺序搜索。当包名使用通配符时这个顺序很重要。这个顺序将决定匹配发生的顺序。这意味着更多特殊的包规范必须在通用的包之前出现。比如，以下动态引用包头信息指定了由 ACME 提供的包的参数：

DynamicImport-Package: *; vendor=acme, *

如果需要使用标识参数动态引用多个包，这个语法允许由分号隔开的包列表，包名称在参数前指定。

在类装载过程，被装载的类所在的包与指定的包名列表比较。每一个匹配的包名反过来将尝试连线到一个导出，这与 Import-Package 规则时一样的。如果一个连线尝试成功了，这个搜索将转发到导出者类加载器，并继续加载类。这个连线在随后不可被更改，即使类没有被装载。这意味着一旦一个包动态解析了，随后对其包的类或资源的加载将作为普通引用处理。

对一个导出语句为了解析一个动态引用包，所有动态引用定义的特性必须与导出语句的特性匹配。所有强制的特性必须在动态引用定义指定且必须匹配。

一旦建立了连接线，任何导出者的约束必须遵循动态引用。

动态引用与可选包非常相似，不过在 Bundle 解析后它们被处理的方法不同。

3.8.3 父代理

Framework 必须总是委托那些以 java 开头的包到父类型装载器。

某些 JVM，包括 SUN 的 JVM，总是做错误的假设——到父类加载器的委托总是发生。这意味着严格层次结构的类加载委托的假设可能导致 NoClassDefFoundErrors。如果 VM 实现总是希望从任意类装载器类查找自己实现的类，且要求从启动类加载器加载不仅限于 java.*的包，这个错误将会发生。

其它必须从启动类装载器加载的包因此可以指定一个系统属性：

org.osgi.framework.bootdelegation

这个属性使用以下格式包含一个列表：

org.osgi.framework.bootdelegation ::= boot-description

(‘,’ boot-description) *

boot-description ::= package-name | (package-name ‘.*’) | ‘*’

这个.*通配符意味着深匹配。那些匹配这个列表的包必须被父类装载器装载。这个 java.*前缀总是默认的，它可以无需指定。

一个通配符意味着 Framework 总是首先委托父类装载器 ，这与 R3 相同。比如，在运行 SUN 的 JVM 时，可能需要指定一个类似以下的值：

org.osgi.framework.bootdelegation=sun.*,com.sun.*

使用这个属性，Framework 必须委托所有的 java.*，sun.*和 com.sun.*包到父类加载器。

3.8.4 搜索顺序全过程

Framework 必须遵循以下规则来加载类或资源。当一个 Bundle 的类加载器需要装载一个类或查找一个资源，搜索必须按以下指定步骤执行：

（1）如果类或资源在 java.*包，这个请求需要委托给父类加载器；否则，继续下一步骤搜索。如果这个请求委托给父类加载器且类或资源不存在，搜索将终止且这个请求导致失败。

（2）如果类或资源来自一个在启动代理列表(org.osgi.framework. bootdelegation)，那么这个请求将委托给父类加载器。如果类或资源没有找到，搜索终止。

（3）如果类或资源在一个使用 Import-Package 定义的引用包或在前一个加载动态引用的包，那么请求将委托给导出这个包的 Bundle 的类加载器；否则搜索将跳到下一步。如果这个请求委托给导出的类加载器且类或资源没有找到，则搜索终止且请求失败。

（4）如果类或资源子一个使用 Require-Bundle 引用一个或多个 Bundle 的包中，这个请求将 Bundle 清单指定的顺序委托到其它 Bundle 的类加载器。这使用一个深度优先策略。在 Bundle 的 Classpath 使用之前搜索所有的 Bundle。如果类或资源没有找到，则搜索将转到下一个步骤。

（5）搜索 Bundle 的内置的 Bundle 的 Classpath。如果类或资源没有找到，这个搜索将转到下一个步骤。

（6）搜索每一个附加的片段 Bundle 的 Classpath。片段按 Bundle ID 升序搜索。如果这个类或资源没有找到，搜索转到下一个步骤。

（7）如果类或资源在一个 Bundle 导出的包或 Bundle 引用的包（使用 Import-Package 或 Require-Bundle），这个搜索将以没有找到指定的类或资源而终止。（说明：如果到这一步说明类或资源没有找到，若这个类或资源包含在刚搜索的 Import-Package 或 Require-Bundle，则说明搜索失败。）

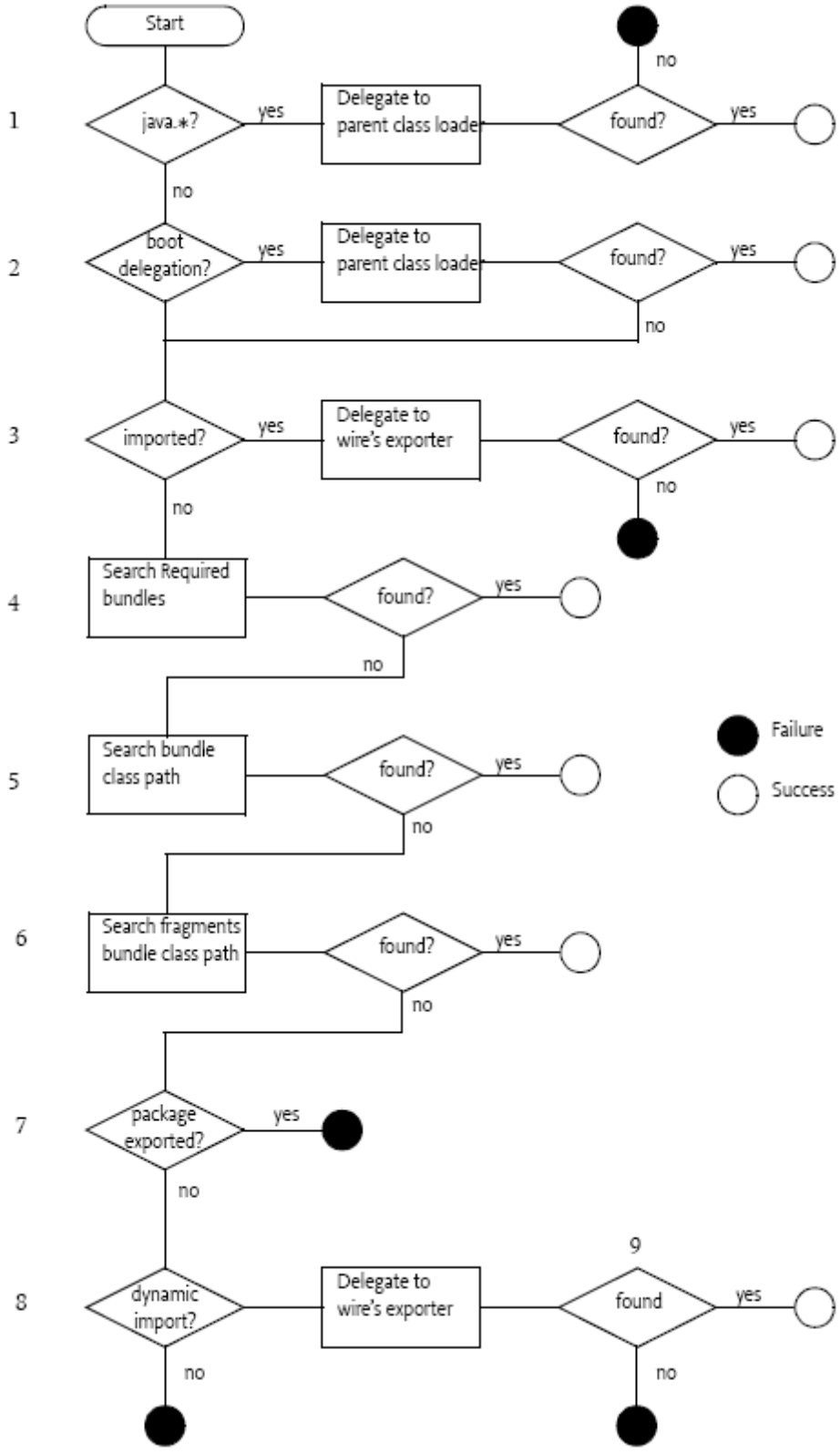
（8）否则，如果类或资源在一个使用 DynamicImport-Package 声明的引用包，那么将尝试动态引用这个包。一个导出者必须遵循任何隐含的约束。如果一个合适的导出者找到了，那么将建立连接线，这样以后的装载将在步骤 3 处理。如果一个动态连接线不能建立，则请求失败。

（9）如果动态引用包的已经被确定，这个请求将委托给导出的 Bundle 的类加载器。如果这个请求被委托给一个导出类加载器且类或资源没有找到，那么搜索终止且请求失败。

当委托给另一个 Bundle 类加载器时，委托请求进入算法的步骤 4.

以下非标转化的流程图演示了描述的搜索过程：

Flow chart for class loading (non-normative)



3.8.5 父类加载器

隐式加载的包是所有 java.*的包，因为这些包是 Java 运行时需要的且不易在同一时刻使用多个版本。比如，所有对象都必须扩展 Object 类。

一个 Bundle 不可以声明 java.*的引用或导出，这样做是一个错误且任何 Bundle 在安装时将失败。所有其它在父类加载器可用的包都必须在执行的 Bundle 中隐藏。

然而，框架必须显式的从父类加载器中导出相关的包。这个系统属性 org.osgi.framework.system.packages 包含了从系统 Bundle 导出的包的描述。这个属性使用标准的 Export-Package 头信息语法：

org.osgi.framework.system.packages ::= package-description (‘,’ package-description) *

在启动路径的一些类假设他们可以使用任何的类加载器来加载其它在启动类路径的类，这对于 Bundle 的类加载器是不正确的。Framework 实现必须尝试从启动类路径加载这些类。

系统 Bundle（ID 为 0）用于暴露非 java.*的包到父类加载器。这个系统 Bundle 的导出定义被当作普通导出处理，这意味着他们可以使用版本信息，且用于作为普通 Bundle 解析过程的方式解析引用定义。其它 Bundle 可能提供对相同包的另一种方法的实现。

父加载器的一些导出定义可以使用属性或 Framework 设置。导出定义必须有指定 Bundle 的特征名称和版本值。

以这种方式父类加载器导出的包必须考虑任何对底层包 uses 的指令。比如，javax.crypto.spec 的定义必须声明它对 javax.crypto.interfaces 和 javax.crypto 的 uses。

3.8.6 资源加载

一个 Bundle 的资源可以通过其类加载器来访问，但它也可以通过 getResource、getEntry 或 findEntries 方法来访问。所有这些方法返回一个 URL 对象或一个 URL 对象的枚举。这些 URL 被称为 Bundle 条目 URLs。这些方法返回的 URLs 的结构可以不同且独立实现。

Bundle 条目 URLs 通常由 Framework 创建，不过，在一些情况，Bundle 需要处理 URL 来查找相关资源。Framework 因此需要确保：

●Bundle 条目 URLs 必须是具有层次结构的。

●作为构建另一个 URL 的上下文。

●Bundle 条目 URL 使用的 java.net.URLStreamHandler 类必须对用于建立 URL 的 java.net.URL 类可用，URL 使用 Framework 定义的结构。

●Bundle 条目 URL 相关的方法 getPath 必须返回 Bundle 的一个资源或条目的绝对路径。比如，getEntry(“myimages/test.gif”)返回的 URL 必须有一个 myimages/test.gif 的路径。

看一下例子，一个类可以使用一个 URL 与一个 index.html 的 Bundle 资源关联并且将资源的 URLs 映射到同 JAR 目录的其它文件。

```
public class BundleResource implements HttpContext {
        URL root; // to index.html in bundle
        URL getResource( String resource ) {
                return new URL( root, resource );
        }
        ...
}
```

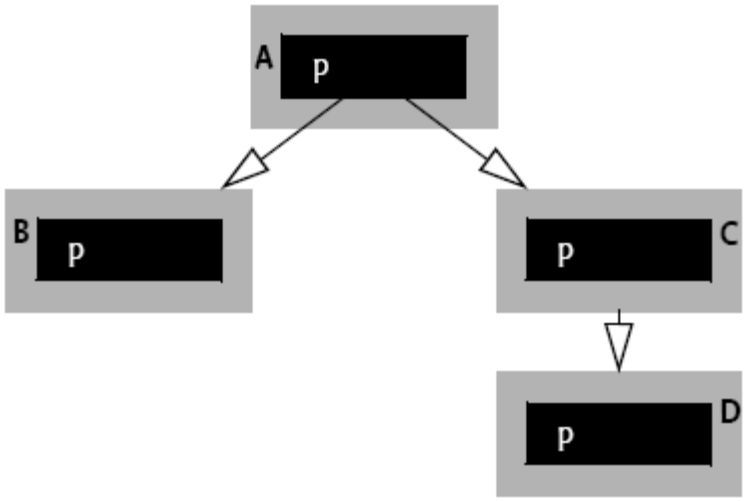
3.8.7 Bundle 循环

多个必需的 Bundle 可以导出相同的包。那些导出相同包的 Bundle，若出现在 Require-Bundle，从这个包搜索类或资源时可能导致循环查询。考虑以下定义：

A: Required-Bundle: B, C

C: Required-Bundle: D

这些定义可用下图描述。

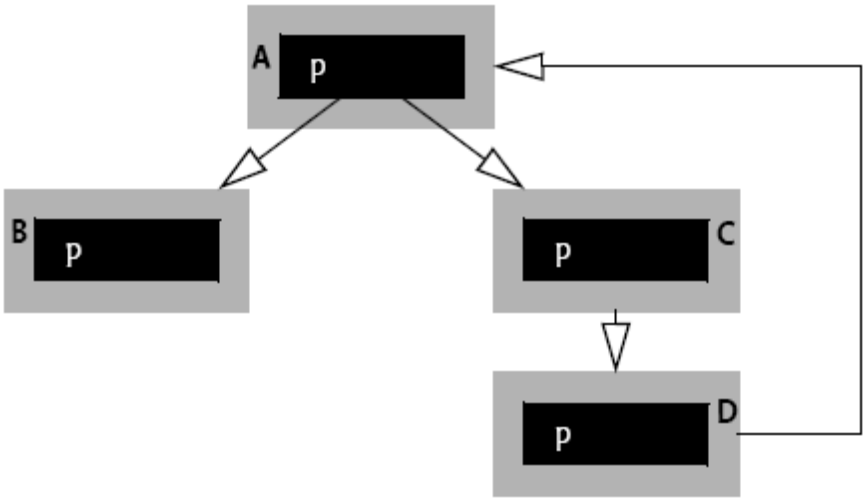


每一个 Bundle 都导出一个包 p。在这个例子，Bundle A 需要 Bundle B、C 和 D。当 Bundle A 装载一个来自 p 的类或资源时，那么必需的 Bundle 的搜索顺序是 B，D，C，A。这是一个深度优先搜索顺序因为需要的 Bundle 在这个 Bundle 的 Classpath 搜索之前搜索（查看

Step 4）。必须的 Bundle 按它们在 Require-Bundler 头信息出现的顺序搜索。如果依赖图有一个环的话，深度优先搜索可能引入无休止的搜索循环。

使用刚才的情景，如果 Bundle D 需要 Bundler A 的话，便引入一个环。

D: Require-Bundler: A



当 Bundle 的类加载器加载一个来自包 p 的类或资源，如果没有考虑循环，那么 Bundle 的搜索顺序可能是：B,B,B,…。

由于当每次到达 Bundle D 时将引入一个环，搜索将会重回到 A 并重新开始。Framework 必须阻止这样的依赖循环来避免无休止的搜索。

为了避免无休止的搜索，Framework 必须在访问它时标记每一个 Bundle 且不再搜索前一个访问过的 Bundle 的必须 Bundle。在以上依赖图使用这种访问标记模式将产生以下搜索顺序：B,D,C,A。

3.8.8 代码在 Started 前执行

只要一个 Bundle 被解析了，那么这个 Bundle 的导出包将暴露给其它 Bundle。这个条件意味着其它 Bundle 可以在 Bundle 导出包启动前调用导出包的方法。

3.9 装载本地代码库（略）

3.10 本地化

一个 Bundle 包含了大量人们可以解读的数据。一些信息可能需要依赖用户的语言、国家和任何指定的参数——如区域具有不同的翻译。这描述了一个 Bundle 如何根据一个区域为清单和其它配置资源提供常用的翻译。

Bundle 本地化条目共享一个通用的基本名称。为了查找一个存在的本地化条目，使用一个下划线（‘_’\u005F）加上一些后缀，通过一个下划线分割并使用.properties 后缀。后缀定义在 java.util.Locale。后缀的顺序必须是：

●语言。

●国家。

●参数。

比如，以下文件提供了将清单翻译到英语、荷兰语（比利时和荷兰）和瑞典。

OSGI-INF/l10n/bundle_en.properties

OSGI-INF/l10n/bundle_nl_BE.properties

OSGI-INF/l10n/bundle_nl_NL.properties

OSGI-INF/l10n/bundle_sv.properties

Framework 通过根据一个指定的区域添加后缀到本地化基本名称，然后搜索本地化条目，并最后加上.properties 后缀。如果一个译文没有找到，一般会首先删除区域变量，然后是国家，最后是语言，直到找到一个包含有效译文的条目。比如，为区域 en_GB_welsh 查找译文将以以下顺序查找：

OSGI-INF/l10n/bundle_en_GB_welsh.properties

OSGI-INF/l10n/bundle_en_GB.properties

OSGI-INF/l10n/bundle_en.properties

OSGI-INF/l10n/bundle.properties

这允许指定更多区域的本地化文件来覆盖更少信息的本地化文件。

3.10.1 查找本地化条目

本地化条目可以包含在 Bundle 或在片段中。因此 Framework 必须首先在 Bundle 查找，然后在它的附加片段中查找。片段 Bundle 必须将本地化条目搜索委托给他的宿主 Bundle 中具有最低 ID 的 Bundle。

Bundle 类加载器不能用来搜索本地化条目。只有搜索 Bundle 的内容和附加片段。本地化条目将从 Bundle 搜索，即使‘.’没有包含在 Bundle 类路径。

3.10.2 清单本地化

本地化值保存在 Bundle 里的 property 资源。Bundle 本地化 property 文件的默认基本名称是 OSGI-INF/l10n/bundle。Bundle-Localization 清单头信息可以用于覆盖本地化文件默认的基本名称。

一个本地化条目包含了本地化信息的 key/value 条目。在 Bundle 清单所有头信息可以被本地化。然而，对于有 Framework 语义的头信息，Framework 必须总是使用非本地化版本。

一个本地化键可以使用以下语法作为一个 Bundle 清单头信息的值指定：

header-value ::= ‘%’text

text ::= < 任何有效的清单头信息值和属性键值 >

比如，考虑到以下 Bundle 清单条目：

Bundle-Name : %acme bundle

Bundle-Vendor : %acme corporation

Bundle-Description : %acme description

Bundle-Activator : com.acme.bundle.Activator

Acme-Defined-Header : %acme special header

用户自定义的头信息也可以被本地化。在本地化键的空格也是允许的。

刚才清单条目的例子可以使用以下的条目来本地化，这个条目在清单本地化条目 OSGI-INF/l10n/bundle.properties 中。

```
# bundle.properties
acme\ bundle=The ACME Bundle
acme\ corporation=The ACME Corporation
acme\ description=The ACME Bundle provides all of the ACME \
services
acme\ special header=user-defined Acme Data
```

以上清单条目也可以在清单本地化条目 OSGI-INF/l10n/ bundle_fr_FR.properties 包含法语本地化。

3.11 Bundle 验证

如果没有指定 Bundle-ManifestVersion，那么 Bundle 清单版本默认为 1，且默写 R4 语法，比如一个新的清单头信息，将被忽略而不是引发一个错误。R3 中 Bundle 必须根据 R3 规范对待。

以下列出了依赖安装 Bundle 失败的错误：

●Bundle-RequireExecutionEnvironment 头信息与可用的执行环境不匹配。

●丢失 Bundle-SymbolicName。

●重复的属性或重复的指令（除非在 Bundle-Nativecode 子句）。

●一个给定包的多个导入。

●导出或引入 java.*。

●使用一个没有定义的 mandatory 属性的导出。

●正在安装一个与已存在的 Bundle 具有相同的特征名称和版本的 Bundle。

●更新一个 Bundle 到一个与另一个 Bundle 具有相同的特征名和版本的 Bundle。

●任何语法错误（比如，不恰当格式的版本、特征名，不认识的指令值，等）。

●在一个包中一起使用不同值指定了 Specification-version 和 version，在清单头信息将视他们相同。比如：

Import-Package p;specification-version=1;version=2 将导致安装失败，但：

Import-Package p;specification-version=1, q;version=2 不会是一个错误。

●清单头信息列出了一个 OSGI-INF/permission.perm，但不存在该文件。

●Bundle-ManifestVesion 值不等于 2，除非 Framework 认识一个最近版本的语义。

3.12 可选

这个规范提供了一个可选的机制：启动类路径扩展。使这个机制可选的原因是不能以一种轻便的方式实现扩展。一个兼容的框架必须依赖启动类路径扩展的实现将以下属性设置为 true 或 false：

●org.osgi.supports.bootclasspath.extension

如果这个属性没有设置或者值不认识，那么这个值默认是 false。一个没有实现类路径扩展的框架必须拒绝安装或更新一个带有这个选项的 Bundle。它在安装或更新时必须抛出一个异常。

此外，框架必须实现片段、要求的 Bundle 和扩展点。他们因此必须设置以下属性为 true。

●org.osgi.supports.framework.requirebundle

●org.osgi.supports.framework.fragments

●org.osgi.supports.framework.extension

3.13 必须的 Bundle

Framework 支持一个机制：Bundle 可以直接连线到其它 Bundle。以下小节定义个了相关头信息并在然后讨论了可能的场景。在最后，将讨论使用 Require-Bundle 的一些结果（有时是没有料想到的）。

3.13.1 Require-Bundle

Require-Bundle 头信息包含了 Bundle 特征名称的名称列表，这些 Bundle 必须在搜索导入后，但在搜索 Bundle 类路径前搜索。片段或扩展 Bundle 不能是要求的。只有由要求的包标记为导出的包才对提出要求的 Bundle 可见。

Require-Bundle 清单头信息必须遵循以下语法：

Require-Bundle ::= bundle-description

(‘,’ bundle-description)*

bundle-description ::= symbolic-name

(‘;’ parameter)*

在 Require-Bundle 头信息中可以使用以下指令：

●visiblity——如果这个值是 private（默认），那么所有来自要求的 Bundle 的可见的包不能被再导出。如果这个值是 reexport，那么需要这个 Bundle 的 Bundles 将可以传递的访问这些导出的包。也就是说，如果 Bundle A 要求 Bundle B，且 Bundle B 要求使用 visibility := reexport 的 Bunle C，那么无论 Bundle A 是否要求 Bundle C，它都可以访问所有 Bundle C 导出的包。

●resolution——如果这个值是 mandatory（默认），那么为了解析这个 Bundle，要求这个 Bundle 必须存在。如果这个值是 optional，Bundle 将依然解析，即使要求的 Bundle 不存在。

以下匹配的属性由 Framework 构建：

●bundle-version——这个属性的值是一个版本范围用于选择要求 Bundle 的 Bundle 版本。默认值是[0.0.0, ∞)。

一个给定的包可能可以从多于一个的要求 Bundle 得到。这样的包被称为“拆分包”，因为他们内容来自不同的 Bundle。如果这些不同 Bundle 提供不可以了的不可预料的相同包，

那么将引起类的重叠，查看“使用要求 Bundle 的问题”节。然而，没有类的重叠的包显然是允许的。

比如，以下为例：

A: Require-Bundle: B

Export-Package: p

B: Export-Package: p; partial=true; mandatory := partial

如果 Bundle C 引入包 p，它将直接连线到包 A.p，然而它的内容将来自 B.p > A.p。Bundle B 的导出定义的 mandatory 属性确保 Bundle B 不是偶然的作为包 p 的导出者。拆分包有一些缺点，将在“使用要求 Bundle 的问题”节讨论。

来自拆分包的资源和类必须以在 Require-Bundle 头信息指定的要求 Bundle 的顺序搜索。

比如，假设一个 Bundle 由一些 Bundle 组成且有一些可选的语言资源（也可以是 Bundle）。

Require-Bundle: com.acme.facade; visibility := reexport,

com.acme.bar.one; visibility := reexport,

com.acme.bar.two; visibility := reexport,

com.acme.bar._nl; visibility := reexport; resolution := optional,

com.acme.bar._en; visibility :=reexport; resolution := optional

一个 Bundle 可能同时引用包和要求一个以上的 Bundle，但如果一个包通过 Import-Package 引用，它还在 Require-Bundle 不可见：Import-Package 将优先于 Require-Bundle，且有一个要求 Bundle 导出的包和通过 Import-Package 引用的不能作为拆分包对待。

为了允许要求一个命名的 Bundle，提出要求的 Bundle 必须有 BundlePermission[<bundle symbolic name>, REQUIRE]权限，其中 Bundle 特征名称是要求的 Bundle 的名称。要求的 Bundle 必须能够提供 Bundle 且因此必须拥有 BundlePermission[<bundle symbolic name>, PROVIDE]权限，其中这个名称指定提出要求的 Bundle。在一个片段 Bundle 要求另一个 Bundle 的情况下，Bundle 必须在片段 Bundle 的保护域里检查。

3.13.2 拆分包兼容性

每当一个包有多个源 Bundle，它就是一个拆分包；只有当使用 Require-Bundle 头信息的 Bundle 才能拥有拆分包。

一个源 Bundle 是提供了给定的包的 Bundle。要求 Bundle 和提出要求的 Bundle 都可以作为源 Bundle。要求的 Bundle 和提出要求的 Bundle 仅可以贡献他们导出的包。

如果一个拆分包的包源是其它源的子集，那么来自两个 Bundle 的导出包列表是兼容的。

3.13.3 使用必须 Bundle 的问题

Bundle 连线的更好的方式是使用 Import-Package 和 Export-Package 头信息，因为他们以更小的程度耦合了导入者和导出者。Bundle 可以重构成拥有不同的包组合，而不会引起其它包失败。

Require-Bundle 头信息提供了一种一个 Bundle 绑定到另一个 Bundle 所有导出的方式，不过导出那些内容。虽然这在第一印象上看起来有点方便，但是它有一些缺点：

●拆分包——使用必须的 Bundle，来自相同包的类可以来自不同的 Bundle，这样的包被称为一个拆分包。拆分包有以下缺点：

A) 完整性——没有一种方式来保证拆分包的所有片段被真正的包含。

B) 次序——如果在多于一个的必须 Bundle 存在相同的类，那么 Require-Bundle 的次序是有意义的。一个错误的次序可以引起难以跟踪的错误，类似传统的 Java 类路径模型。

C) 性能——当包被拆分，一个类必须在所有提供商搜索。当必须抛出 ClassNotFoundException（需要很多代价）时这增加了大量 的时间。

D) 混乱——很容易找到一个有潜在混乱的安装。比如，以下安装是没有直觉的：

A: Export-Package: p; uses := q

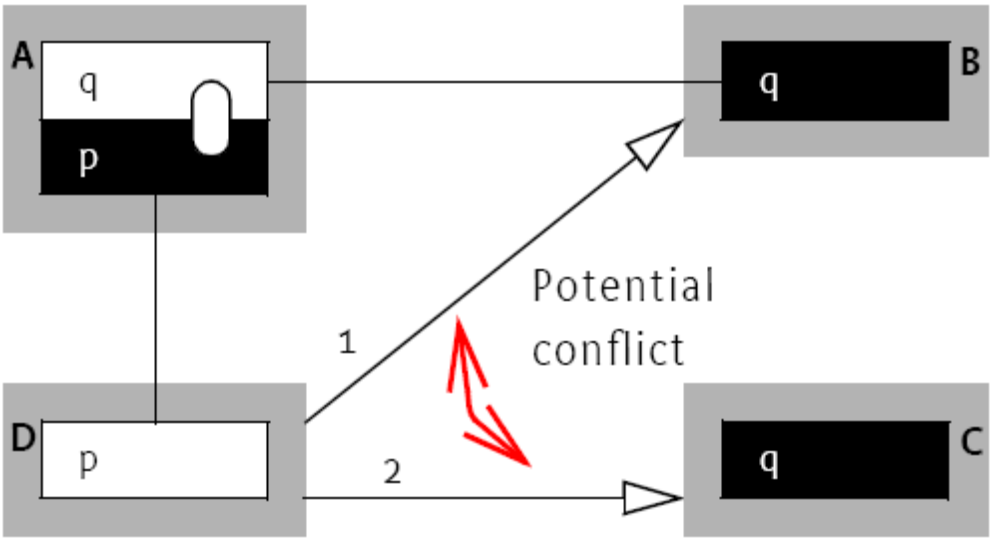
Import-Package: q

B: Export-Package: q

C: Export-Package: q

D: Require-Bundle: B, C

Import-Package: p



在这个示例，Bundle D 合并来自 Bundle B 和 C 的拆分包 q，然而，在 Bundle A 对 p 的导出使用了来自包 q 的约束。这意味着 Bundle D 可以看到来自 Bundle B 可用的包 q，但不能看到来自 Bundle C 的无效 q。这个连线是允许的因为在大多数情况下这没有什么问题。然而，在极少数情况下，当包 C.q 包含了一些在 B.q 也包含的类时一致性可能被打破。

●不稳定导出——visibility := reexport 功能，必须包的导出标记可以根据必须 Bundle 的导出标记而发生意外的改变。

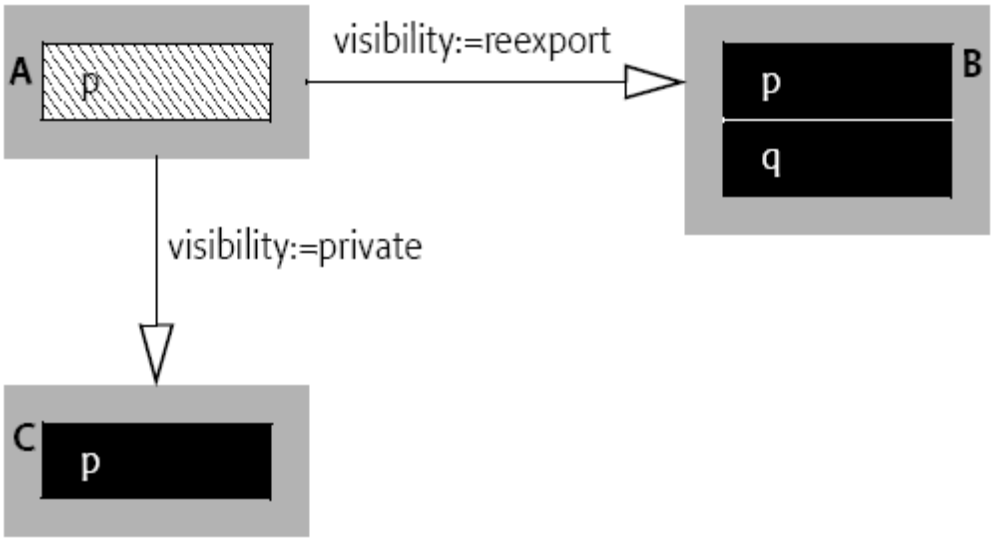
●重叠——在提出要求的 Bundle 的类会根据必须 Bundle 的导出标记和必须 Bundle 包含的类与必须 Bundle 发生重叠。（相反的，Import-Package，除非使用 resolution := optional，整个包的重叠类与导出者无关。）

●意外的标记变更——Require-Bundle 指令 visibility := private（默认）可能会在一些如以下示例展示的环境发生意外的覆盖。

A: p (private, not exposed in manifest)
Require-Bundle: B; visibility := reexport,
C; visibility := private
B: Export-Package: p
C: Export-Package: p

Bundle 的导出标记仅包 p 组成。然而，包 p 是拆分的。Framework 将在 Bundle B，然后是 C，且最后是 A 来搜索一个类。

因此在必须 Bundle C 的 visibility := private 指令没有与包 p 相关的任何影响。然而，如果 Bundle B 被更改为停止导出包 p，那么这个指令将起作用且包 p 将从 Bundle A 的导出标记中被删除。这在下图给出演示。



3.14 片段 Bundle

片段是由框架附加到宿主 Bundle 的 Bundle。附加作为解析的一部分完成：框架在宿主解析之前添加片段 Bundle 的相关定义到宿主定义。片段因为作为宿主的一部分对待，包括任何允许的头信息；他们不可以拥有类加载器。片段必须拥有自己的保护域。

片段的一个关键场景是为不同的区域提供译文文件。这允许译文文件被独立于主应用 Bundle 对待和组装。

当一个片段 Bundle 被更新，前一个片段必须依然附加到它的宿主 Bundle。更新的片段的新内容不允许附加到宿主 Bundle 直到框架重启或宿主 Bundle 被刷新。在这一期间，一个附加片段有两个版本：老的版本，附加的宿主的老版本且新的片段 Bundle 可以附加到一个新的版本或不同的宿主 Bundle。

在这种情况，包管理服务必须只返回提供的 Bundle 的最新信息。在先前描述的情况，getHosts 方法必须返回片段 Bundle 的新版本的宿主 Bundle，且 getFragments 方法必须返回附加到新版本宿主 Bundle 的片段 Bundle。

当附加一个片段 Bundle 到一个宿主 Bundle，框架必须执行以下步骤：

1） 将片段 Bundle 的导入定义添加到宿主 Bundle，这些定义不能与宿主的导入定义冲突。一个片段可以为宿主私有包提供一个导入语句。在那种情况在宿主的私有包被隐藏。

2） 附加片段的 Require-Bundle 条目到宿主 Bundle 的 Require-Bundle，这些条目不能与宿主的 Require-Bundle 条目冲突。

3） 附加片段 Bundle 的导出定义到宿主导出定义，除非相同的定义（指令和属性必须匹配）已经在宿主存在。片段 Bundle 可以因此为相同的包名添加导出。bundle-version 属性和 bundle-symbolic-name 属性将影响宿主 Bundle。

当宿主和片段 Bundle 不能提供一致类空间类解析，他们便发生了冲突。如果发现一个冲突，片段 Bundle 不能附加到宿主 Bundle。

当一个片段 Bundle 已经附加到它的宿主 Bundle，它必须进入到 resolved 状态。

在运行时，片段的 JAR 在宿主类路径之后被搜索，正如“运行时片段”描述。

一个片段 Bundle 不能被另一个 Bundle 使用 Require-Bundle 头信息要求。

3.14.1 片段宿主

一个片段是附加到一个称为宿主 Bundle 的 Bundle 的 Bundle。片段的组件，像 Bundle-Classpath 和其它定义，添加到宿主 Bundle 定义的最后。在 Export-Package 头信息情况，Bundle 依赖属性如 bundle-version 和 bundle-symbolic-name 来自宿主。在片段的所有类和资源必须使用宿主 Bundle 的类加载器加载。

Fragment-Host 清单头信息必须遵循以下语法：

Fragment-Host ::= bundle-description

bundle-description ::= symbolic-name

(‘;’ parameter)*

以下指令由框架为 Fragment-Host 定义：

●extension——指定这个扩展是一个系统或启动类路径。只有当宿主 Bundle 是系统 Bundle 时才可以使用。这在“扩展 Bundle”节讨论。它支持以下值：

A) framework——片段 Bundle 是一个框架扩展 Bundle。

B) bootclasspath——片段 Bundle 是一个启动类路径扩展 Bundle。

片段必须是指定系统 Bundle 或 system.bundle 别名的实现的 Bundle 特征名称。当 Bundle 特征名称没有引用到系统 Bundle 时框架应该使扩展 Bundle 安装失败。

以下特性由框架为 Fragment-Host 提供：

●bundle-version——选择提供宿主 Bundle 的版本范围。默认值是[0.0.0, ∞)。

当一个片段 Bundle 已经解析，框架必须附加片段 Bundle 到使用最高版本的选择的宿主 Bundle。当一个片段 Bundle 被加到它的宿主 Bundle，它从逻辑上变成其一部分。片段 Bundle 的所有的类和资源必须使用它的宿主 Bundle 的类加载器加载。一个宿主 Bundle 的片段 Bundles 必须以片段 Bundle 安装的顺序附加到宿主 Bundle，片段安装的顺序是以 Bundle ID 的

升序。如果在片段 Bundle 的附加过程中发生了一个错误，那么这个片段不能附加到宿主。当一个片段 Bundle 已经成功附加到它的宿主 Bundle 时它的状态必须进入到 resolved。

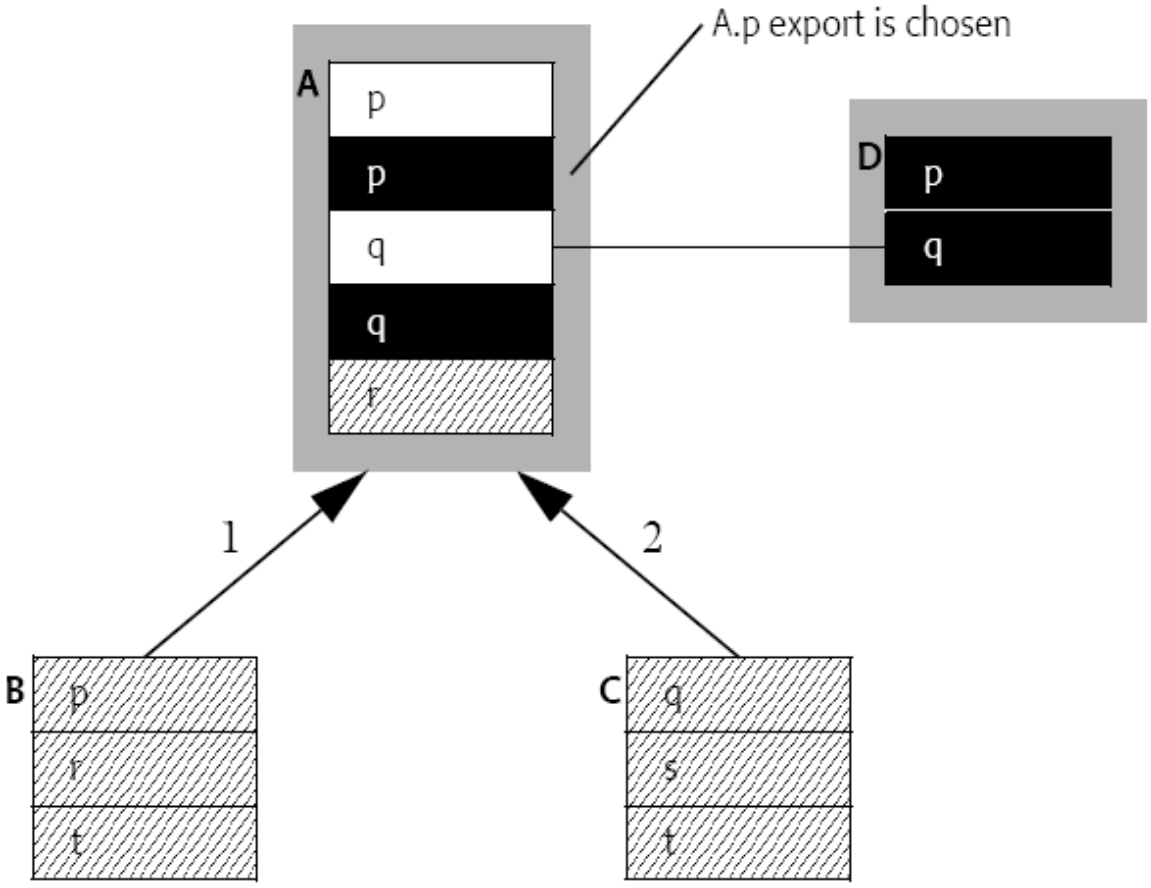
为了使一个宿主 Bundle 允许附加片段，宿主 Bundle 必须有 BundlePermisson[<bundle symbolic name>, HOST]权限。为了被允许附件到宿主 Bundle，一个片段 Bundle 必须有 BundlePermission[<bundle symbolic name>, FRAGMENT]权限。

3.14.2 运行时片段

一个片段的所有类或资源加载通过宿主类加载器处理，一个片段永远都不能有自己的类加载器。片段 Bundle 被看作他们宿主的固有的一部分。

虽然一个片段 Bundle 不能有自己的类加载器，当它不是一个扩展 Bundle，它依然必须有一个分开的保护域。每一个片段可以有自己的权限，这些权限可以连接到片段 Bundle 位置和签名者。

一个宿主 Bundle 的类路径在一个片段类路径搜索之前搜索。这意味着包可以通过宿主和任何它的片段拆分。搜索片段必须以 Bundle ID 升序进行。这是片段 Bundle 被安装的顺序。



上图展示了一个使用两个片段的安装。Bundle B 在 Bundle C 之前安装，且这两个 Bundle 都附加到 Bundle A。下表展示了在这个安装产生的不同的包。需要注意添加到附加的顺序（>）是重要的。

请求的包	来自	备注
p	A.p > B.p	Bundle A 导出了包 p，因此，它将为 p 搜索它的类路径。这个类路径由 JAR 组成且然后是它的片段 Bundles。
q	D.q	引用没有处理拆分包且包 q 从 Bundle D 引用，因此，C.q 无法找到。
r	A.r > B.r	包 r 没有引入且因此来自类路径。
s	C.s	
t	B.t > C.t	

在以上的例子，如果包 p 已经从 Bundle D 被引入，这个表将会看起来有点不同。包 p 将来自 Bundle D，且 Bundle A 自己的内容和 Bundle B 的内容将被忽略。

如果包 q 有 Bundle D，那么类路径会被搜索，且 A.q 将有 A.q > C.q 组成。

片段必须保持附加状态，只要宿主依然被解析。当一个宿主 Bundle 变为 unresolved，那么所有的片段必须从宿主片段分离。当一个片段 Bundle 编程未解析，框架必须：

●从宿主分离。

●重新解析宿主 Bundle。

●重新附加已经附加的 Bundle。

一个片段可以通过在它自己或宿主上调用 refreshPackages 方法或 resolveBundle 方法变成未解析。

3.14.3 片段 Bundle 无效的头信息

以下列表包含了不能在一个片段 Bundle 使用的头信息：

●Bundle-Activator

3.15 扩展的 Bundle

扩展 Bundle 可以是框架实现的可选部分或提供必须附属启动类路径的功能。这些包不能通过一般的引用/导出机制来提供。

启动类路径扩展是必要的，因为某些包实现假定他们在启动类路径或他们被所有客户端要求使用。一个启动类路径扩展的例子是一个想 JSR 169 的 java.sql 的实现。启动类路径扩展不要求被兼容的框架实现，查看“可选”一节。

框架扩展是必要的，它用于提供框架实现的一些方面。比如，一个框架卖主可以使用框架扩展 Bundle 提供可选的服务，像权限管理服务和启动级别服务。

一个扩展 Bundle 应该使用系统 Bundle 实现的 Bundle 特征名称，或者它可以使用系统 Bundle 的别名——system.bundle。

一下例子使用 Fragment-Host 清单头信息来为一个特定的框架实现指定一个扩展 Bundle。

Fragment-Host: com.acme.impl.framework; extension := framework

以下例子使用 Fragment-Host 清单头信息来指定一个启动类路径扩展 Bundle。

Fragment-Host: system.bundle; extension := bootclasspath

以下描述了一个扩展 Bundle 的生命周期：

1） 当一个扩展 Bundle 被安装，它进入到 INSTALLED 状态。

2） 在框架裁决时，这可能需要要求一个框架重启动，扩展 Bundle 允许进入 RESOLVED 状态。

3） 如果扩展 Bundle 被刷新，那么框架必须关闭，宿主 VM 必须终止，且框架必须重启动。

4） 如果一个 RESOLVED 扩展 Bundle 被刷新，那么框架必须关闭，宿主 VM 必须终止且框架必须重新运行。

5） 当一个 RESOLVED 扩展 Bundle 被更新或 UNINSTALLED，它不允许再次进入 RESOLVED 状态。如果扩展 Bundle 刷新了，那么框架必须关闭，宿主 VM 必须终止且框架必须被重新运行。

更新一个扩展 Bundle 到另一个类型的 Bundle 是允许的。如果旧的扩展 Bundle 被解析了，那么它必须作为系统 Bundle 的片段附加。当这个 Bundle 被更新了，Bundle 的旧内容必须保持附加到系统 Bundle 直到系统 Bundle 被刷新或扩展 Bundle 被刷新（使用包管理服务）。这必须初始化一个 VM 且框架将重启。当框架启动，Bundle 的新内容必须被解析。

所有扩展 Bundle 的 Bundle 事件应该与普通 Bundle 一样被派发。

3.15.1 扩展 Bundle 无效的清单头信息

如果一个扩展 Bundle 被安装或更新且它指定了以下头信息，那么必须抛出一个 BundleException 异常。

- Import-Package。
- Require-Bundle。
- Bundle-NativeCode。

●DynamicImport-Package。

●Bundle-Activator。

启动类路径和框架扩展 Bundle 允许指定一个 Export-Package 头信息。任何由框架扩展 Bundle 导出的包，当扩展 Bundle 被解析时必须由系统 Bundle 导出。

3.15.2 类路径处理

一个启动类路径扩展 Bundle 的 JAR 文件必须被添加到宿主 VM 的启动类路径。一个框架扩展 Bundle 的 JAR 被添加到框架的类路径。

扩展 Bundle 必须以扩展 Bundle 安装的顺序被添加到他们的类路径，也就是说，以 Bundle ID 升序。

一个框架来配置它自己或启动路径来添加扩展 Bundle 的 JAR 实现上很特殊。在一些执行环境，它可能无法支持扩展 Bundle。在这样的环境，当一个扩展 Bundle 被安装，框架必须抛出一个 BundleException。Bundle 异常的结果必须有一个 UnsupportedOperationException 类型的原因。

3.16 安全（略）

3.17 R4.1 变更（略）

3.18 参考资料（略）

4 生命周期层

4.1 介绍

生命周期层提供了用于来控制 Bundle 的安全和生命周期操作的 API。这个层次基于 Module 和安全层。

4.1.1 要点

●完整——生命周期层必须实现一个涵盖 Bundle 的安装、启动、停止、更新、卸载和监控的 API。

●反射——API 必须能够精确洞察 Framework 状态。

●安全——它必须能够在使用细粒度控制的权限的安全环境下使用 API。然而，安全是可选的。

●管理——它可能需要远程的管理服务平台。

4.1.2 词汇

●Bundle——表示 Framework 中一个安装的 Bundle。

●Bundle Context——在 Framework 中一个 Bundle 的执行上下文。当 Bundle 被启动或停止时 Framework 传递上下文到一个 Bundle 激活类（Activator）。

●Bundle Activator——在 Bundle 的一个类实现的接口，用于启动或停止 Bundle 这个 Bundle。

●Bundle Event——对 Bundle 生命周期操作的通知事件。这个事件通过 Bundle Listener 来接收。

●Framework Event——在异常或 Framework 状态改变后发生的事件。这个事件通过 Framework Listener 接收。

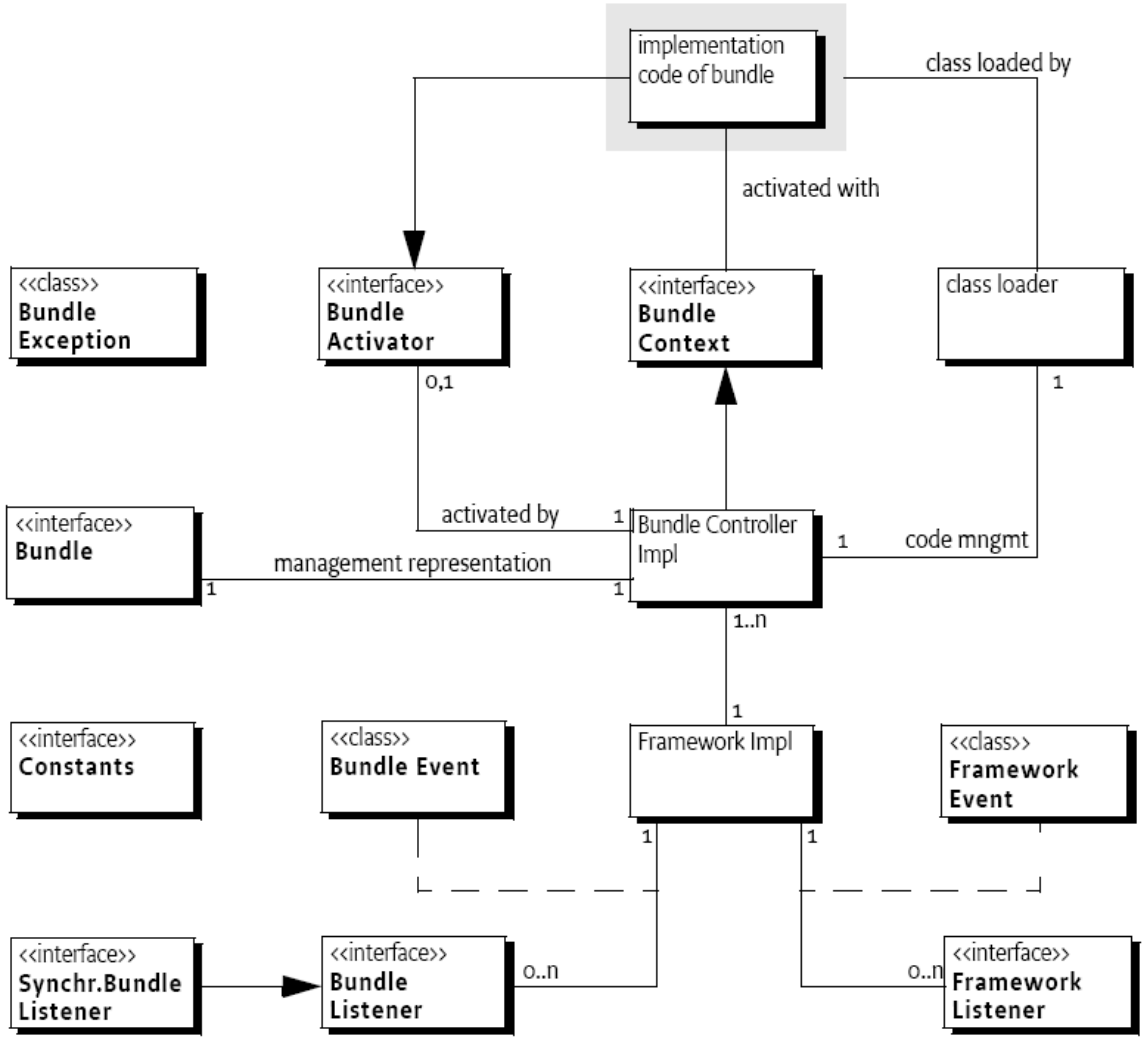
●Bundle Listener——Bundle 事件监听器。

●Synchronous Bundle Listener——同步传输 Bundle 事件的监听器。

●Framework Listener——Framework 事件监听器。

●Bundle Exception——当 Framework 操作失败时抛出的异常。

●System Bundle——一个表示 Framework 的 Bundle。



4.2 Bundles

一个 Bundle 表示一个在 OSGi 框架执行的 JAR 文件。Bundle 的类加载方面在 Module 层描述了。然而，Module 层没有定义如何安装、更新和卸载一个 Bundle。这些生命周期的操作在这章定义。

一个 Bundle 的安装只能通过另一个 Bundle 或通过实现特殊的方法（比如，作为一个 Framework 实现的命令行参数）。

一个 Bundle 通过它的激活器来启动。它的激活器在 Bundle-Activator 清单中定义。给定的类必须事先 BundleActivator 接口。这个接口有一个 start 和 stop 方法，Bundle 使用它们来注册监听器和启动必须的线程等。Stop 方法必须完成资源的清理并终止任何运行的线程。

在 Bundle 的激活器上，它接收一个 Bundle 上下文。Bundle 上下文接口的方法可以被大致分为以下几类：

●信息——访问 Framework 其它的信息。

●生命周期——可能安装其它 Bundle。

●服务注册表——服务注册表将在服务层中讨论。

4.3 Bundle 对象

对于在 OSGi 服务平台安装的每一个 Bundle，都有一个关联的 Bundle 对象。一个 Bundle 的 Bundle 对象用于管理 Bundle 的生命周期。这通常由一个管理 Agent 来完成，这个管理 Agent 同时也是一个 Bundle 对象。

4.3.1 Bundle 标识

一个 Bundle 由一些名字来标识，并根据他们使用范围发生变化：

●Bundle 标识（Identifier）——一个长整型的整数，在一个 Bundle 的整个生命周期中，由 Framework 赋值的唯一标识，及时它被更新或 Framework 重启了。它的目的是区别在一个 Framework 中的 Bundle。当 Bundle 被安装时，Bundle 标识按升序标识给 Bundle。方法 getBundleID()用于返回 Bundle 的标识。

●Bundle 位置（Location）——一个由管理 Agent（Operator）在安装 Bundle 时赋值给它的名字。这个字符创通常表示为一个到一个 JAR 文件的 URL，不过这不是强制的。在一个特殊的 Framework，一个 Location 必须是唯一的。一个位置字符串唯一标识 Bundle 且在更新 Bundle 后不能被改变。方法 getLocation()用于取回一个 Bundle 的位置。

●Bundle 特征名称（Symbolic Name）——由开发人员赋值的名称。Bundle 版本和特征名称的组合是一个 Bundle 的全局唯一标识。方法 getSymbolicName()用于返回其特征名称。

4.3.2 Bundle 状态

一个 Bundle 可以是以下状态的一种：

●INSTALLED——Bundle 已经成功安装。

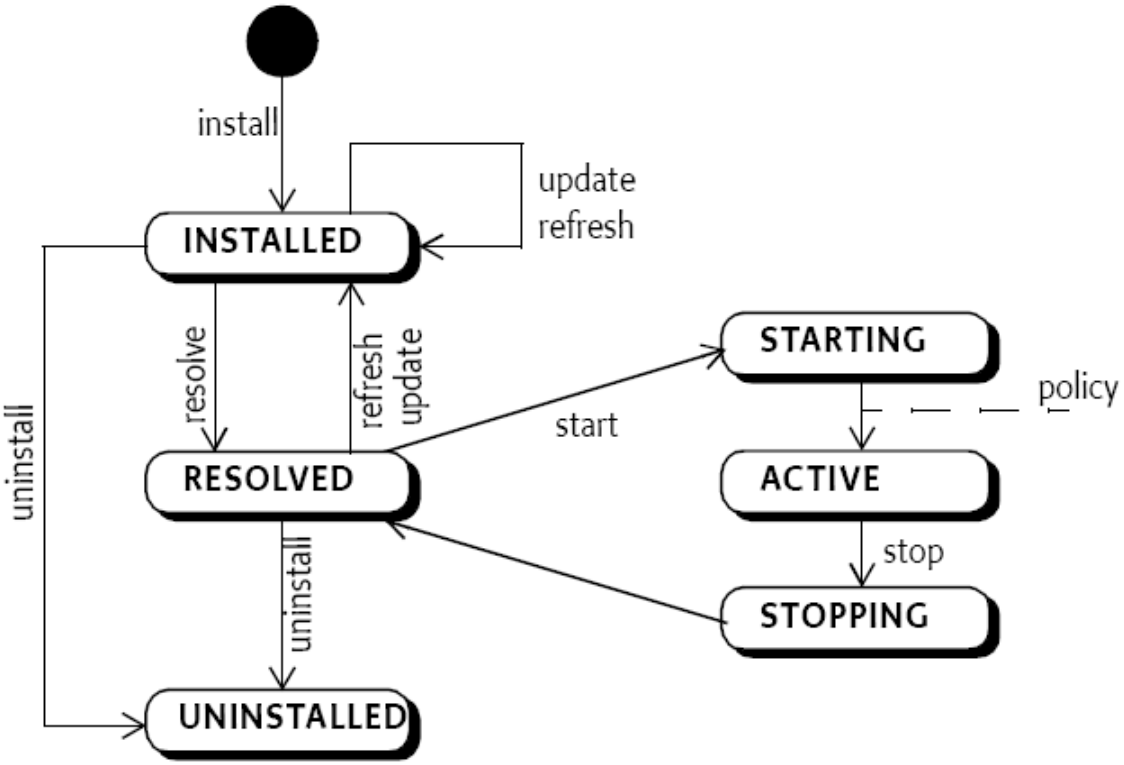
●RESOLVED——Bundle 需要的所有 Java 类已经可用。这个状态意味着 Bundle 准备启动或者已经停止。

●STARTING——Bundle 正在被启动，BundleActivator.start 方法将被调用且这个方法还没有返回。当 Bundle 有一个激活政策，这个 Bundle 将维持在 STARTING 状态，直到根据它的激活政策来激活它。

●ACTIVE——Bundle 已经被成功激活且正在运行，它的 Bundle 激活器的 start 方法已经被调用且成功返回。

●STOPPING——Bundle 正在被停止。BundleActivator.stop 方法已经被调用但还没有返回。

●UNINSTALLED——Bundle 已经被卸载，它不能迁移到另一个状态。



当一个 Bundle 被安装了，它被存储在 Framework 的持久存储中并一直保存在那直到它被显式的卸载。一个 Bundle 是否启动或停止必须记录在 Framework 的持久存储。一个已经在启动后被持久记录的 Bundle 必须启动，不管 Framework 什么时候启动，直到被显示的停止。Start Level 服务影响 Bundle 的实际启动和停止。

Bundle 接口定义个了一个 getState()方法用于返回 Bundle 的状态。

如果这个规范使用 active 名词来描述一个状态，那么它包括 STARTING 和 STOPPING 状态。

Bundle 状态用掩码来表示，不过一个 Bundle 在任何时刻都只能有且只有一个。以下代码实例用于决定一个 Bundle 是否在 STARTING、ACTIVE 或 STOPPING 状态。

```
if ((b.getState() & (STARTING | ACTIVE | STOPPING) != 0)
        doActive()
```

4.3.3 安装 Bundles

在 Bundle Activator 给定的 BundleContext 接口定义了以下方法来安装一个 Bundle：

●installBundle(String)——从指定的 URL 位置安装一个 Bundle。

●installBundle(String,InputStream)——使用指定的输入流对象安装一个 Bundle。

一个 Bundle 在安装前必须通过验证，否则安装必须失败。Bundle 的验证在 Bundle 验证小节讨论。

每一个 Bundle 由位置字符串唯一标识。如果一个 Bundle 使用了指定的位置，installBundle 方法必须为已安装的 Bundle 返回一个 Bundle 对象，而不是新建一个 Bundle 对象。

Framework 必须给 Bundle 赋值一个唯一标识，这个唯一个标识的值比已有的标识要大。

在 Framework 安装一个 Bundle 必须是：

●持久的——Bundle 必须跨 Framework 和 JVM 保持安装状态直到被卸载。

●原子性的——安装方法必须完成 Bundle 的安装，或者如果安装失败了，OSGi 服务平台必须将它保持在调用这个方法之前的那个状态。

一旦一个 Bundle 被安装了，一个 Bundle 对象被创建且其它生命周期操作必须在这个对象上执行。返回的 Bundle 对象可以用来 start、stop、update 和 uninstall 这个 Bundle。

4.3.4 解析 Bundle

当 Framework 成功解析了在清单描述的 Bundle 依赖关系后，这个 Bundle 将迁移到 RESOLVED 状态。

4.3.5 启动 Bundle

一个 Bundle 可以通过调用 Bundle 对象的任一 start 方法来启动或如果这个 Bundle 被指定了 ready 和 autostart 设置来指定它必须被启动，那么 Framework 能够自动启动这个 Bundle。

只有当以下条件满足了，一个 Bundle 才是 ready：

●这个 Bundle 已经被解析了依赖。

●如果使用可选的 Start Level 服务，那么 Bundle 的启动等级被满足了。

一旦一个 Bundle 被启动了，这个 Bundle 必须被激活，以给予控制权使它能够初始化。激活可以立即发生（Eager Activation），或者在第一个类被加载时激活（Lazy Activation）。在一个重启或更改启动级别后，一个已启动的 Bundle 可能需要自动的再次被 Framework 启动。Framework 因此需要为每一个 Bundle 持久一个 autostart 设置。这个 autostart 设置可以是以下值：

●Stopper——这个 Bundle 不该被启动。

●Started with eager activation——一旦这个 Bundle 已经 ready，她必须被启动，然后立即激活。

●Started with declared activation——一个这个 Bundle 已经 ready，它必须被启动，然后根据它声明的激活策略来激活。

Bundle 接口定义了 start(int)方法来启动一个 Bundle 且控制其 autostart 设置。Start(int)方法使用一个整形选项，这个选型定义了以下值：

●0——使用立即激活启动并设置 autostart 设置为 Started with eager activation。如果这个 Bundle 已经使用懒激活策略启动并正在等待启动，那么它必须立即被启动。

●START_TRANSIENT——同 0 行为相同，不过它不更改 autostart 设置。如果 Bundle 不能被启动，比如，一个 Bundle 还没有 ready，那么必须抛出一个 Bundle 异常。

●START_ACTIVATION_POLICY——使用清单头信息 Bundle-ActivationPolicy 声明的激活策略启动 Bundle，然后设置 autostart 为 Started with declared activation。

●START_ACTIVATION_POLICY|START_TRANSIENT——使用 Bundle 声明的激活策略启动但不修改 autostart 设置。

当 Framework 尝试启动 Bundle 时，如果它还没有解析依赖，必须尝试解析 Bundle。如果解析失败，start 方法必须抛出一个 BundleException。在这种情况下，Bundle 的 autostart 设置必须被设置，除非使用 START_TRANSIENT。

当 start 方法没有异常返回时，Bundle 状态将是 ACTIVE 或 STARTING，这根据声明的激活策略并是否使用来决定。

start()方法调用 start(0)。

可选的 Start Level 服务将影响 Bundle 的实际的启动顺序和停止顺序。这个服务也可以用来查询 autostart 设置：

●isBundlePersistentlyStarted(Bundle)——如果 Bundle 的 autostart 设置为 Stopped 则返回 false，其它情况均返回 true。

●isBundleActivationPolicyUsed(Policy)——如果 Bundle 的 autostart 设置指定必须使用清单头文件的激活策略，返回 true，如果 Bundle 必须被立即激活返回 false。

片段 Bundle 不能被启动，如果尝试启动它必须抛出一个 Bundle 异常。

4.3.6 激活

一个 Bundle 若存在激活器，则通过调用它的 Bundle 激活器来激活。BundleActivator 接口定义了 Framework 启动和停止 Bundle 时调用的方法。

为了告知 OSGi 环境作为 Bundle 激活器的类的全名，Bundle 开发人员必须在 Bundle 清单文件的 Bundle-Activator 头信息中声明激活器。Framework 必须初始化一个该类的对象并将它转换为一个 BundleActivator 实例。它然后必须调用 BundleActivator.start 方法来启动 Bundle。

以下是 Bundle-Activator 清单信息的例子：

Bundle-Activator : com.acme.Activator

一个作为 Bundle 激活器的类必须实现 BundleActivator 接口，它被声明为 public 且有一个 public 的默认构造器，因此它的实例可以使用 Class.newInstance 来创建。

提供 Bundle 激活器是可选的。必须，一个仅到处一些包的类库 Bundle 不需要定义一个 Bundle 激活器。此外，其它来控制和获取 Bundle 上下文的机制，比如服务组件运行时（Service Component Runtime）。

BundleActivator 接口定义了启动和停止一个 Bundle 的方法：

●start(BundleContext)——这个方法可以用来分配 Bundle 需要的资源、启动线程、注册服务等。如果这个方法没有注册任何服务，Bundle 可以在以后需要时注册，比如，在一个回调或外部事件，主要它在 ACTIVE 状态就行。如果这个方法抛出异常，Framework 必须标记 Bundle 为停止并发送 STOPPING 和 STOPPED 时间，但它必须调用 Bundle 激活器的 stop(BundleContext)方法。

●stop(BundleContext)——这个方法必须撤销在 BundleActivator.start (BundleContext)发生的所有动作。然而，卸载服务或者卸载 Framework 事件侦听器并不是必须的，因为他们无论如何都会被 Framework 清理掉。

一个 Bundle 激活器在 Bundle 启动时必须被创建，也隐含着创建一个类加载器。对于大型系统，贪婪策略会明显增加启动时间且不必要增加内存开销。如服务组件运行时和激活策略机制可以减轻这些问题。

片段 Bundle 不可以指定一个 Bundle 激活器。

4.3.7 激活政策

一个 Bundle 的激活也可以在使用激活政策的启动后指定。这个政策在 Bundle-ActivationPolicy 头信息指定，它具有如下语法：

Bundle-ActivationPolicy ::= policy (‘;’ directive) *

Policy ::= ‘lazy’

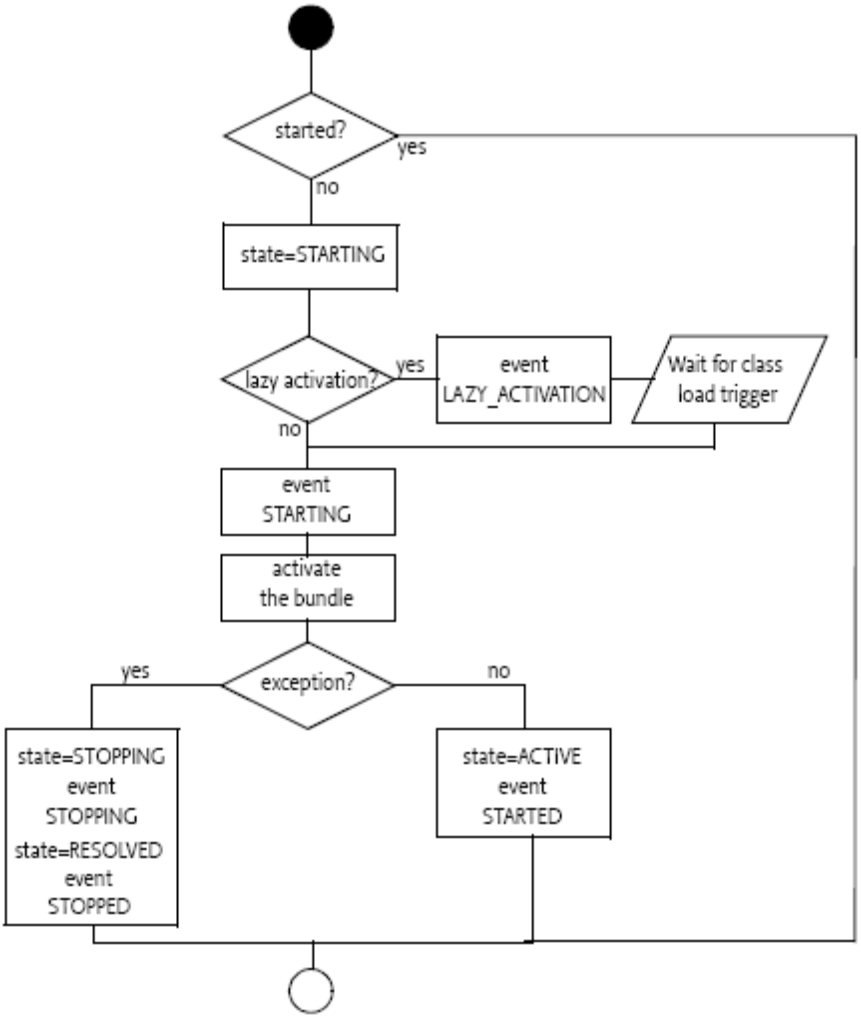
它唯一定义的政策是晚激活政策。如果没有指定这个头信息，Bundle 将使用立即激活政策。

4.3.7.1 晚激活政策

一个晚激活政策指定了一个 Bundle，一旦被启动，它不会被激活，直到从该 Bundle 加载一个类，或者在正常的类加载或通过 Bundle 的 loadClass 方法。加载资源不会触发激活。与默认的里激活政策相比，它的变化反应在 Bundle 的状态和事件。当一个 Bundle 使用晚激活政策启动时，将需要完成以下步骤：

- 为这个 Bundle 创建一个 Bundle 上下文。
- Bundle 的状态迁移到 STARTING 状态。
- 触发 LAZY_ACTIVATION 事件。
- 系统等待从该 Bundle 一个类装载发生。
- 触发正常的 STARTING 事件。
- Bundle 被激活。
- Bundle 状态迁移到 ACTIVE 状态。
- 触发 STARTED 事件。

如果因为 Bundle 激活器的 start 方法抛出异常导致激活失败，Bundle 必须无须调用 Bundle 激活器 stop 方法而停止。这些步骤如以下流程图描述。这个流程图还显示了正常立即激活和晚激活政策的不同。



晚激活政策允许一个 Framework 实现来延迟 Bundle 类加载器的创建和 Bundle 的激活，直到 Bundle 第一个被使用，内在的在启动时节约了资源和启动时间。

默认的，任何类加载都可以触发晚激活，然而，资源加载不能触发激活。晚激活政策可以使用以下指令定义那些类引起激活。

●include——一个包名的列表，当一个类从这些包加载时，必须触发激活。默认是 Bundle 的所有包。

●exclude——一个包名列表，一个类从这些包加载时，不能触发激活。默认是没有任何包。

比如：

Bundle-ActivationPolicy: lazy;

include ::= “com.acme.service.base, com.acme.service.help”

当一个类加载触发晚激活，Framework 必须首先定义触发类。这个定义能够触发额外的晚激活。这些激活必须被延迟直到所有过渡的类的加载和定义已经完成。在那之后，激活必须按发现的反序执行。也就是说，最晚检测到的激活必须最先执行。只有所有延迟的激活完成了类的加载，激活才完成，此加载必须触发激活并返回加载的类。如果在这个过程中出现错误，它必须作为一个 Framework 的 ERROR 事件被报告。然而，类加载必须正常完成。一个使用晚激活政策激活失败的 Bundle，它不应该被触发激活，直到 Framework 重启或显示的调用 Bundle 的 start 方法。

4.3.8 停止 Bundle

Bundle 接口定义了 stop(int)方法，用于停止一个 Bundle。这将停止一个 Bundle 并将状态设置为 RESOLVED。stop(int)方法是用了一个整型参数。以下是这个值得定义：

●0——如果这个 Bundle 被激活，那么它将冻结这个 Bundle 并设置 autostart 为 Stopped。

●STOP_TRANSIENT——如果这个 Bundle 被激活，那么冻结这个 Bundle。它不改变这个 Bundle 的 autostart 设置。

stop()方法调用 stop(0)。

可选的 Start Level 服务影响 Bundle 实际的启动和停止。

如果试图停止一个片段 Bundle，则必须触发一个 Bundle 异常。

4.3.9 冻结

BundleActivator 接口定义了一个 stop(BundleContext)方法，由 Framework 调用来停止一个 Bundle。这个方法必须释放在激活分配的资源。跟正在停止的 Bundle 相关的线程必须被立即停止。一旦这个 stop 方法返回，线程的代码不能再使用 Framework 相关的对象。

如果正在停止的 Bundle 在它的生命周期已经注册了一些服务，那么 Framework 在 stop 方法必须自动卸载所有的服务。

Framework 必须保证如果一个 BundleActivator.start 方法被成功执行了，当 Bundle 被冻结时，必须调用相同的 BundleActivator 对象的 BundleActivator.stop 方法。调用 stop 方法后，这个特定的 BundleActivator 再也不能被使用了。

由一个停止的 Bundle 导出的包对其它 Bundle 仍然可用。这种不间断的导出意味着其它 Bundle 可以执行来自一个停止的 Bundle 的代码，且 Bundle 设计者必须确保这种情况是无害的。当 Bundle 没有被启动时，导出接口是唯一一种阻止这种不当的执行的方式。通常情况下，为了确保他们不被执行，接口不应该包含可执行代码。

4.3.10 更新 Bundle

Bundle 接口定义了两个方法用于更新一个 Bundle：

●update()——这个方法更新一个 Bundle。

●update(InputStream)——这个方法用指定的 InputStream 对象更新一个 Bundle。

更新过程支持从 Bundle 的一个版本迁移到同一个 Bundle 的更新版本。一个更新的 Bundle 的导出必须对 Framework 立即生效。如果不使用任何旧的导出，那么旧的导出必须被删除。否则，所有旧的到处必须对存在的 Bundle 保留可用且直到调用了 refreshPackages 方法或 Framework 重启后以后才使用它解析。

对安装的 Bundle 或新的 Bundle，它的更新器 必须有一个 AdminPermission [<bundle>, LIFECYCLE]，AdminPermission 参数在管理权限中解释。

4.3.11 卸载 Bundle

Bundle 接口定义了 uninstall()方法用于从 Framework 卸载一个 Bundle。这个方法会引起 Framework 对其它 Bundle 通知该 Bundle 正在被卸载，且设置 Bundle 的状态为 UNINSTALLED。无论何种情况，Framework 都必须删除跟这个 Bundle 相关的资源。这个方法必须总是从 Framework 的持久存储中卸载 Bundle。

一旦这个方法返回，OSGi 服务平台的状态必须与这个 Bundle 没有被安装的状态一样，除非：

●卸载的 Bundle 有导出包。

●卸载的包被 Framework 选择作为这些包的导出者。

如果不使用任何旧的导出，那么旧的导出必须被删除。否则，所有旧的到处必须对存在的 Bundle 保留可用且直到调用了 refreshPackages 方法或 Framework 重启后以后才使用它解析。

4.3.12 检查 Bundle 的变更

Bundle 对象提供了一个检查变更的便利方法。Framework 必须谨记一个 Bundle 将被任何生命周期操作而改变。方法 getLastModified()将返回 Bundle 被安装、更新或卸载的最后一个时机。最后更改时机必须被持久存储。

这个方法将返回相对 1970 年 1 月 1 日 UTC 时间的毫秒数。

当一个 Bundle 正在从另一个 Bundle 缓存资源且在 Bundle 更改时需要刷新缓存时非常有用。当缓存 Bundle 没有激活时，这些目标 Bundle 的生命周期更改可以发生。最后更改时间因此是一个跟踪这些 Bundle 的便利方法。

4.3.13 取出清单头信息

Bundle 接口定义了两个方法来取出清单头信息：getHeaders()和 getHeaders(String)。

●getHeaders()——返回一个 Dictionary 对象，它包含了清单头和值序列对。这些值是根据 java.util.Locale.getDefault 返回的默认区域返回本地化的值。

●getHeaders(String)——返回一个 Dictionary 对象，它包含了清单头和值序列对。返回值使用指定的区域本地化。区域可以使用以下值：

A) null——使用由 java.util.Locale.getDefault 返回的默认区域。这与 getHeaders 返回值相同。

B) 空串——返回清单原始头信息，不进行本地化。包含任何“%”开头的头信息。

C) 指定的区域——给定的区域用于本地化清单头信息。

本地化根据“本地化”描述的执行。如果对于一个 key，没有找到相应的译文，那么由 Bundle.getHeaders 返回的 Dictionary 将返回在清单指定的原始值，但去除了“%”开头的值。

这些方法需要 AdminPermission[<bundle>, METADATA]，因为一个清单头信息可能是敏感的，比如在 Export-Package 头信息定义的包列表。Bundle 总是有权限访问它们自己的头信息。

在 Bundle 进入 UNINSTALLED 状态后，方法 getHeaders 必须继续提供清单头信息。一个 Bundle 被卸载之后，这个方法仅返回原始清单信息或 Bundle 卸载时默认的区域本地化的值。

一个 Framework 实现在处理清单头信息时，必须只适用原始清单信息。本地化不能影响 Framework 的操作。

4.3.14 装载类

在某些情况下，我们需要装载一些类，就好象从 Bundle 内部装载似的。loadClass(String)方法用于访问 Bundle 的类加载器。这个方法可以在以下地方被使用：

●从另一个 Bundle 加载插件。

●启动应用系统模型激活器。

●与遗留下载的代码交互。

比如，一个应用系统模型可以使用这个方法来加载 Bundle 的初始化类请根据应用系统模型来启动它。

```
void appStart() {
        Class initializer = bundle.loadClass(activator);
        if ( initializer != null ) {
                App app = (App) initializer.newInstance();
                app.activate();
        }
}
```

从一个 Bundle 加载一个类，如果这个 Bundle 使用晚加载政策，它可引起激活。

4.3.15 资源访问

一个 Bundle 的资源可以来自不同的地方。他们可以来自原始 JAR 文件、片段 Bundle、导入包，或者 Bundle 的类路径。不同的情况需要不同的资源搜索策略。Bundle 接口提供了一些访问资源的方法，这些方法使用不同策略访问资源。以下搜索策略是支持的：

●类空间(Class Space)——方法 getResource(String)和 getResources(String)提供了与类空间一致的资源的访问，这与 3.8.4 描述的一致。遵循这种搜索顺序可能是某些 JAR 文件不可达。这些方法要求 Bundle 被解析了。如果 Bundle 没有被解析，Framework 将尝试去解析它。

搜索顺序可以隐藏某些 JAR 文件的目录。分开的包也被考虑到了；因此，使用相同包名的资源可以来自不同的 JAR 文件。如果 Bundle 没有被解析，getResource 和 getResources 方法必须仅从 Bundle 类路径加载。这个搜索策略应该被想要访问自己资源的代码使用。调用任何一个方法可以引起类加载器的创建并强迫解析 Bundle。

●JAR 文件——方法 getEntry(String)和 getEntryPaths(String)提供了在 Bundle 的 JAR 文件的资源的访问。不需要任何搜索，仅考虑原始的 JAR 文件。这个方法的目的是提供一个无需考虑 Bundle 是否解析的低级访问。

●Bundle 空间——方法 findEntries(String,String,Boolean)是一个中间格式。当需要来自另一个 Bundle 的配置或安装信息时使用。它考虑片段 Bundle，但他从不创建类加载器。这个方法提供了对关联 JAR 文件的所有目录的访问。

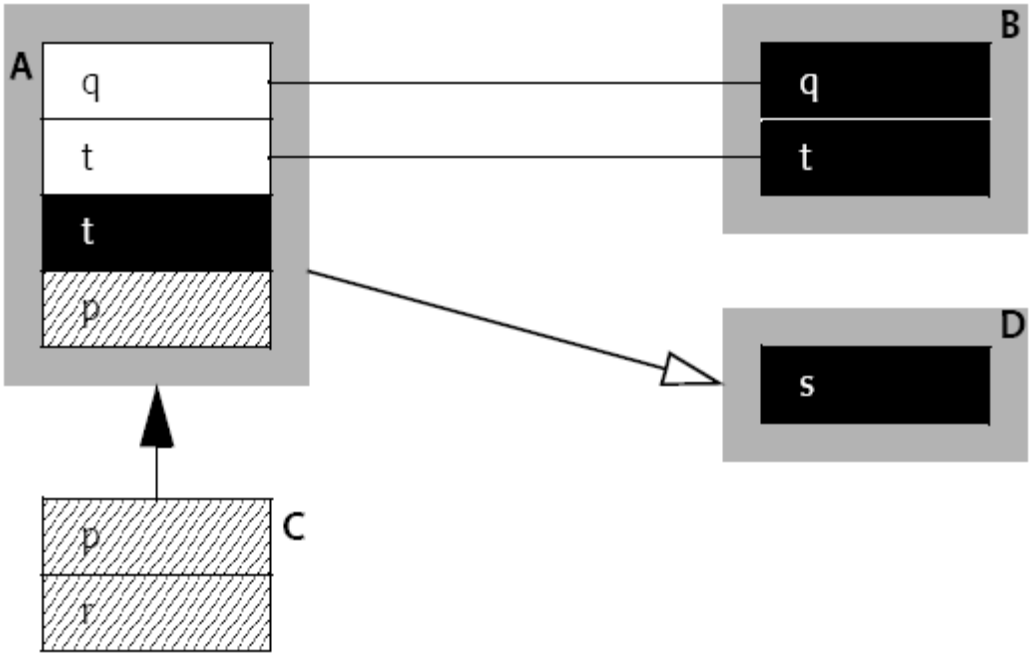
如下示例，考虑到以下安装：

```
A: Require-Bundle: D
   Import-Package: q,t
   Export-Package: t
B: Export-Package: q,t
```

C: Fragment-Host: A

D: Export-Package: s

对应的连接图如下：



下表显示了当 A 被解析后从这个安装获取一个资源的对比。

资源	getResource	getEntry	findEntries
q	B.q	null	null
p	A.p > C.p	A.p	A.p > C.p
r	C.r	null	C.r
s	D.s	null	null
t	B.t	A.t	A.t

下表显示了当 A 没有被解析时，从其获取一个资源的对比。

资源	getResource	getEntry	findEntries
q	null	null	null
p	A.p	A.p	A.p
r	null	null	null
s	null	null	null
t	A.t	A.t	A.t

4.3.16 Bundle 的权限

Bundle 接口定义了一个方法用于返回 Bundle 权限的附属信息：hasPermission(Object)。如果 Bundle 的保护域有指定的权限，则返回 true，且如果没有或者如果参数指定的对象不是 java.security.Permission 的实例，返回 false。参数类型为 Object，因此 Framework 可以在一个不支持基于 Java2 安全的 Java 平台来实现。

4.3.17 Bundle 的 Bundle 上下文的访问

已经被启动的 Bundle 都具有一个 Bundle 上下文。这个对象是一个容器；它本意是只能被 Bundle 使用。然而，有一些情况下，Bundle 必须作为其它 Bundle 的代表。比如，服务组件运行时作为其它 Bundle 代表来注册服务。Framework 因此提供通过 getBundleContext()方法提供对另一个 Bundle 上下文访问。如果那个 Bundle 没有 Bundle 上下文，那是因为这个 Bundle 是一个片段 Bundle 或那个 Bundle 的状态不在 STARTING、ACTIVE 或 STOPPING，那么它将返回 null。

这个方法具有潜在危险性，因为它允许任何 Bundle 代表其它 Bundle 操作。在安全系统，这个方法需要被 AdminPermission[*,CONTEXT]保护。

4.4 Bundle 上下文

Framework 和它安装的 Bundle 的关系通过使用 BundleContext 对象来关联。一个 BundleContext 对象表示在 OSGi 服务平台一个 Bundle 的执行上下文，且作为对底层 Framework 的代理。

一个 BundleContext 对象在 Bundle 启动时被 Framework 创建。Bundle 可以使用这个私有的 BundleContext 对象来：

- 安装新 Bundle 到 OSGi 环境。
- 查询安装到 OSGi 环境的其它 Bundle。
- 获取持久存储区域。
- 取出已注册服务的服务对象。
- 在 Framework 服务注册表注册服务。
- 订阅和取消订阅 Framework 广播的事件。

当一个 Bundle 被启动了，Framework 创建一个 BundleContext 对象，并将这个对象作为参数传到 Bundle 激活器的 start(BundleContext)方法。每一个 Bundle 由 Framework 提供这个

Bundle 自己的 BundleContext 对象；这些对象不应该在 Bundle 间传递，因为 BundleContext 对象关系到一个 Bundle 的安全和资源定位。

当 stop(BundleContext)方法返回后，这个 BundleContext 对象不能再被使用。Framework 实现必须抛出异常，如果在 Bundle 停止后还是用 BundleContext 对象的话。

4.4.1 获取 Bundle 信息

这个 BundleContext 接口定义了获取关于在 OSGi 服务平台安装的 Bundle 的信息。

●getBundle()——获取跟 BundleContext 关联的 Bundle 对象。

●getBundles()——获取 Framework 安装的 Bundle 数组。

●getBundle(long)——通过指定 Bundle ID 来获取指定的 Bundle。

Bundle 访问没有限制；任何 Bundle 可以枚举安装的 Bundle。然而，那些可以标识一个 Bundle 的信息，比如本地化或头信息，只能提供给具有 AdminPermission[<bundle>,METADATA]的调用者。

4.4.2 持久存储

Framework 必须为平台安装的每一个 Bundle，使用相同文件系统支持，提供一个私有的持久存储。

BundleContext 接口使用 File 类定义了对这个存储的访问，它支持平台独立的文件和目录命名的定义。

BundleContext 接口定义了访问私有存储区域方法：getDataFile(String)。这个方法使用一个相对文件名作为参数。它能够将这个文件名转换为 Bundle 持久存储区域的绝对文件名。然后返回一个 File 对象。如果不支持持久存储，它将返回 null。

Framework 必须自动提供 FilePermission[<storage area>, READ | WRITE | DELETE]来允许 Bundle 读、写和删除存储区域的文件。

如果需要 EXECUTE 权限，那么可以使用文件权限定义的相对路径名。比如，FilePermission[bin/*,EXECUTE]指定了 Bundle 私有数据区域的子目录可能包含可执行文件。这仅提供在 Java 环境的执行权限且不能处理可执行文件相关的底层操作系统问题。

这种特殊处理仅用在 Bundle 的 FilePermission 对象。默认权限不可以接受这种特殊处理。通过 setDefaultPermission 方法设置的相对路径名称的一个 FilePermission 必须被忽略。

4.4.3 环境属性（表略）

BundleContext 定义了返回 Framework 属性的方法：getProperty(String)。这个方法用于返回以下 Framework 属性：

所有 Framework 属性可能被操作员定义为系统属性。如果这些属性没有被定义为系统属性，那么 Framework 必须根据相关标准 Java 系统属性来构建这些属性。

属性名	描述
org.osgi.framework.version	Framework 规范版本，必须是 1.3。
org.osgi.framework.vendor	Framework 实现的提供商。
org.osgi.framework.language	使用的语言。
org.osgi.framework.<< executionenvironment	一个逗号隔开的执行环境列表。
org.osgi.framework.processor	
org.osgi.framework.os.version	
org.osgi.framework.os.name	
org.osgi.supports.<< framework.extension	
org.osgi.supports.<< bootclasspath.extension	
org.osgi.supports.<< framework.fragment	
org.osgi.supports.<< framework.requirebundle	
org.osgi.supports.<< bootdelegation	
org.osgi.supports.<< system.packages	

4.5 系统 Bundle

除了普通 Bundle，Framework 本身也作为 Bundle。代表 Framework 的 Bundle 被定义为系统 Bundle。通过系统 Bundle，Framework 可以注册服务来被其它 Bundle 使用。这些服务的例子是包管理和权限管理服务。

系统 Bundle 通过 BundleContext.getBundles()来列出来，虽然它和其它 Bundle 具有以下不同之处：

●系统 Bundle 总是被赋予为 0 的标识。

●系统 Bundle 的 getLocation()方法返回在 Constants 定义的“System Bundle”字符串。

●系统 Bundle 对于特殊的版本有一个唯一的 Bundle 特征名。然而，system.bundle 名称必须作为实现定义名称标识。

●系统 Bundle 生命周期不能像普通 Bundle 来管理。它的生命周期方法必须按以下方式：

A) start——不需要任何操作，因为系统 Bundle 已经被启动了。

B) stop——立刻返回并启动另一个线程关闭 Framework。

C) update——立即返回并启动另一个线程停止和重启 Framework。

D) uninstall——Framework 必须抛出一个 BundleException 异常来指示系统 Bundle 不能被卸载。

E) 查看系统启动和关闭来获取有关 Framework 启动和停止的更多信息。

●系统 Bundle 的 Bundle.getHeaders 方法返回一个词典对象，它包含实现指定清单头信息。比如，系统 Bundle 清单头信息应该包含一个 Export-Package 头信息声明那些由 Framework 导出的包（比如，org.osgi.framework）。

4.6 事件

OSGi 框架生命周期层支持以下类型的事件：

●BundleEvent——报告 Bundle 生命周期内的变化。

●FrameworkEvent——报告 Framework 启动、启动级别变化、刷新包或者遇到一个错误。

报告的实际事件可以使用 getType 方法获得。从这个方法返回的整数可以是在类里描述的常量中的一个。然而，在将来事件可以且将被扩展。不认识的事件应该被忽略。

4.6.1 监听器

一个监听器接口关联每一种类型的事件。以下描述了这些监听器。

●BundleListener 和 SynchronousBundleListener——当一个 Bundle 生命周期已经被更改后，使用 BundleEvent 的事件类型调用。

SynchonousBundleListener 对象在事件处理过程中同步的调用且必须在任何 BundleListener 对象被调用前调用。当 Bundle 移到另一个状态后，Framework 将发送以下事件：

A) INSTALLED——在一个 Bundle 安装后发送。现在这个 Bundle 的状态是 INSTALLED 状态。

B) RESOLVED——当 Bundle 被解析后由 Framework 发送。现在这个 Bundle 的状态是 RESOLVED。

C) LAZY_ACTIVATION——Bundle 已经指定一个激活政策，它的激活被推迟到一定时间。这个状态被设置到 STARTING 状态。这仅被发送给 SynchronousBundleListener 对象。

D) STARTING——当 Framework 即将激活一个 Bundle 时发送。这只发送给 SynchronousBundleListener 对象。这个状态现在是 STARTING 状态。

E) STARTED——当 Framework 已经激活一个 Bundle 发送该事件。现在 Bundle 状态是 ACTIVE。

F) STOPPING——当 Framework 即将停止一个 Bundle 或 Bundle 激活器的 start 方法已经抛出一个异常且 Bundle 被停止时设置。这个事件表示 Bundle 上下文即将销毁。这个事件只能发送给 SynchronousBundleListener 对象。

G) STOPPED——当 Framework 已经停止一个 Bundle 后设置。

H) UNINSTALLED——当 Framework 已经卸载一个 Bundle 后设置。

I) UNRESOLVED——当 Framework 检测到一个 Bundle 无法解析后设置；这只能在 Bundle 被刷新或更新时发生。当使用管理 API 刷新一些 Bundle 然后这些 Bundle 的每一个都必须有一个 UNRESOLVED 的 Bundle 事件发布。UNRESOLVED 的 Bundle 事件必须在这些 Bundle 的所有 Bundle 都已停止后发布而且若使用一个同步的 Bundle 监听器的情况下，必须在这些 Bundle 的任何一个 Bundle 重新启动之前。RESOLVED 和 UNRESOLVED 并不成对。

J) UPDATED——当一个 Bundle 被更新后发送。

●FrameworkListener——使用一个 FrameworkEvent 的事件类型被调用。Framework 时间有以下类型：

A) ERROR——需要在一个操作后立即受关注的错误。

B) INFO——某些情况下感兴趣的信息。

C) PACKAGES_REFRESHED——Framework 已经刷新包。

D) STARTED——Framework 已经完成所有初始化且以正常方式运行。

E) STARTLEVEL_CHANGED——当设置一个新的启动级别后发送。

F) WARNING——对一个操作的警告，这个操作不是关键的但可能会是一个潜在的错误。

BundleContext 接口方法定义了那些可以用来添加和删除每一个类型的监听器。

事件可以被异步传输，除非另行描述，这意味着他们不需要在产生事件同一个线程传输。用于调用事件监听器的线程没有被定义。

Framework 必须发布一个 FrameworkEvent.ERROR，如果对一个时间监听器的回调引发一个没有检查的异常——除非当回调发生时正在传输一个 FrameworkEvent.ERROR（阻止一个无限循环）。

4.6.2 传输事件

如果框架传输一个异步事件，它必须：

●收集在发布事件的时候的监听器的列表的快照，而不是在事件传输之前，这使得在事件发生之后无法再添加监听器列表。

●确保，在抓快照的时候，列表的监听器属于激活的 Bundle 的时候事件才被传输。

●可能使用超过一个线程传输事件。如果是这种情况那么每一个处理句柄必须以事件发布相同的顺序接收到这些事件。这确保了处理句柄以期望的方式看到事件的发生。

如果 Framework 在当事件发布时没有去捕捉监听器列表，而不是等到事件传输前才捕捉，那么可能发生以下错误：一个 Bundle 可能已经启动且注册了一个监听器，且然后 Bundle 可以查看到它自己的 BundleEvent.INSTALLED 事件。

以下 3 个场景演示了这个概念。

1） 1 个事件场景：

●事件 A 发布。

●监听器 l 注册。

●尝试对事件 A 的异步传输。

期待的行为：监听器 l 一定不能接收到事件 A，因为它在事件发布时并没有被注册。

2）2 个事件场景：

●监听器 2 注册。

●事件 B 发布。

●监听器 2 卸载。

●尝试事件 B 的异步传输。

期待行为：监听器 2 接收到事件 B，因为监听器在事件发布时已经被注册。

3）3 个事件场景：

●监听器 3 被注册。

●发布事件 C。

●注册监听器 3 的 Bundle 被停止。

●尝试事件 C 的异步传输。

期待行为：监听器 3 一定不能接收到事件 C，因为它的 Bundle 上下文对象已经无效。

4.6.3 异步缺陷

通常的，一个调用一个监听器的 Bundle 不应该保留任何 Java 线程监控器。这意味着当初始化一个回调时，不管是 Framework 还是一个同步事件的原始发起者都不应该由线程监控器监控。

Java 线程监控器的目的是保护数据的更新。这必须是很小的代码段，它们不调用任何代码，这些代码的行为不能被监视。从 synchronized 代码调用 OSGi 框架可能引起意料不到的副作用。这些副作用的其中之一可能是死锁。一个死锁在两个线程因为互相等待对方而阻塞的情况下发生。

超时检测能够用来打破死锁，但 Java 线程监控器没有超时检测。因此，这些代码会一直挂起知道系统被重启。死锁可以通过避免同步调用 Framework 来阻止。

如果当调用其它代码必须加锁的话，使用 Java 线程监控器来创建信号量，这个信号量可以用于超时检测且因此提供一个走出死锁情况的机会。

4.7 框架启动和关闭

一个框架的实现必须在提供任何服务之前启动。操作人员如何启动 Framework 在该规范没有详细描述，因为它因不同实现而不同。一些框架的实现可能提供命令行操作，而其它人可能从一个配置文件读取启动信息。在所有情况，框架实现必须按给定的顺序执行以下操作。

4.7.1 启动

当启动一个框架，必须执行以下活动：

1）启动事件处理。那么现在事件可以传输给监听器。

2）系统 Bundle 进入 STARTING 状态。

3）所有先前被标记为启动的安装 Bundle 必须按 Bundle.start 方法描述的启动。在启动中，任何异常必须被包装成一个 BundleException 且然后作为一个 FrameworkEvent.ERROR 事件类型的框架事件发布。Bundle 和他们的状态在 Bundle 对象这一节讨论了。如果框架实现了可选的启动级别规范，这个行为会有所不同。查看“启动级别服务规范”一节。任何指定一个激活政策的 Bundle 必须根据激活政策处理。

4）系统 Bundle 进入 ACTIVE 状态。

5）广播 FrameworkEvent.STARTED 的框架事件。

4.7.2 关闭

框架偶尔也需要关闭。关闭也可以通过通知系统 Bundle 来初始化，这在“系统 Bundle”节包含了。当框架被关闭，必须按顺序完成以下操作：

1）系统 Bundle 进入 STOPPING 状态。

2）所有激活的 Bundle 按照“Bundle.stop 方法”描述的关闭，除了他们持久记忆的状态指示他们必须在框架下一次启动时重启，它们将保持不变。在关闭时抛出的任何异常必须包装成一个 BundleException 且作为 FrameworkEvent.ERROR 类型的事件发布。如果框架实现了可选的启动级别规范，这个行为会不同。在关闭期间，使用晚激活政策的 Bundle 一定不能被激活，及时从他们那加载了类。

3）事件处理被管理。

4.8 安全

4.8.1 管理权限

4.8.2 使用签名

4.8.3 回调权限

4.8.4 晚激活

4.9 变更

4.10 参考资料

5 服务层

5.1 介绍

OSGi 服务层定义了一个动态协作模型，它与生命周期层紧密关联。服务模型是一个发布、查找和绑定的模型。一个服务是一个普通的 Java 对象，服务使用服务注册表注册为一个或多个 Java 接口。Bundle 可以注册服务、搜索服务或在注册状态更改时接收通知。

5.1.1 要点

●协作——服务层必须提供一个 Bundle 发布、查找和绑定到其它 Bundle 的服务，而不需要了解其它 Bundle 的机制。

●动态——服务机制必须能够处理外部变化和底层结构变化。

●安全——必须提供对服务访问的限制。

●自描述——提供服务层内部状态的完全访问。

●版本标记——提供一个机制来使其能够处理 Bundle 和服务随着时间推移发生的变化。

●持久标识——提供一个跨框架重启跟踪服务的方法。

5.1.2 词汇

●服务——以一个或多个接口和属性注册在服务注册表的对象。这个对象可以被 Bundle 发现和使用。

●服务注册表——保存服务注册。

●服务引用——对一个服务的引用。提供服务属性的访问而不是真正的服务对象。服务对象必须能够通过一个 Bundle 的 Bundle 上下文查询。

●服务注册——当服务注册时提供的收据（返回值）。服务注册允许更新服务的属性和卸载服务。

●服务权限——在注册或使用一个服务时，使用一个接口名称的权限。

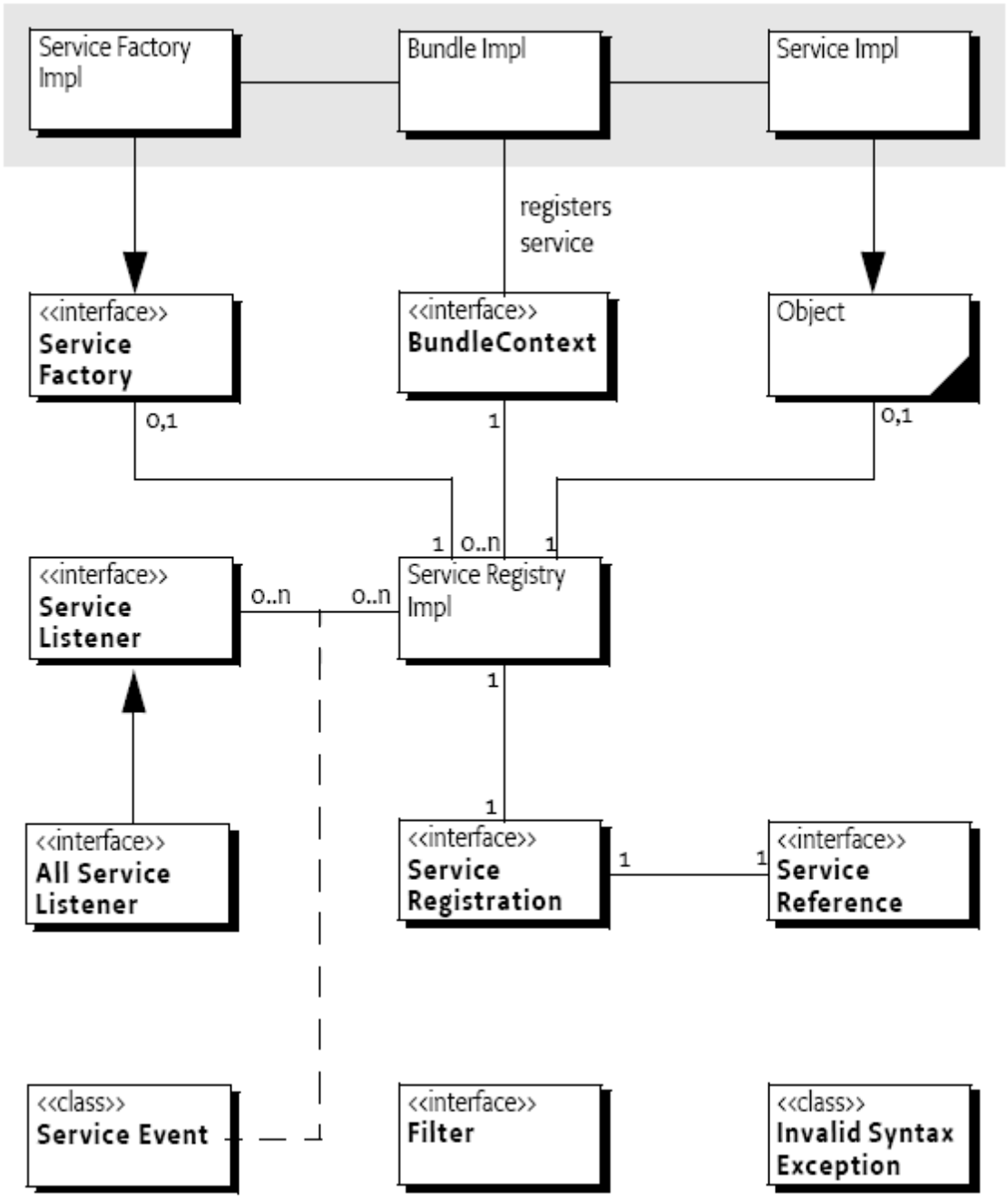
●服务工厂——一个服务类，可以使正在注册的 Bunlde 可以为每一个正在使用的 Bundle 自定义服务对象。

●服务监听器——服务事件的监听器。

●服务事件——保存关于注册、更改或卸载一个服务对象的信息的事件。

●过滤器——一个实现了简单但强大的过滤语言的对象。它可以选择属性。

●无效语法异常——当一个过滤表达式包含一个错误时抛出的异常。



5.2 服务

在 OSGi 服务平台，Bundle 围绕着一些互相合作的服务来被构建，这些服务来自一个共享的服务注册表。这样的一个 OSGi 服务在语义上通过它的作为服务对象服务接口和实现来定义。

服务接口应该尽可能少的指定实现细节。OSGi 为公共需求指定了很多服务接口且在将来会指定更多。

服务对象属于一个 Bundle 并在一个 Bundle 内运行。这个 Bundle 必须使用框架服务注册表来注册服务对象，使得框架控制的其它 Bundle 能够使用服务的功能。

拥有服务的 Bundle 和使用服务的 Bundle 之间的依赖由 Framework 管理。比如，当一个 Bundle 停止了，由这个 Bundle 使用 Framework 注册的所有服务都必须自动的卸载。

Framework 将服务映射到底层服务对象且提供一个简单和强大的查询机制，使得一个 Bundle 可以查询到它需要的服务。Framework 同时还提供了事件机制，使得 Bundle 可以接收到注册、更改和卸载服务的事件。

5.2.1 服务引用

一般来讲，注册的服务通过 ServiceReference 对象来引用。这避免了，当一个 Bundle 需要知道一个服务但不需要服务对象本身，创建没有必要的 Bundle 间动态服务依赖。

一个 ServiceReference 对象可以没有包含任何依赖在 Bundle 间存储或传递。当一个 Bundle 希望使用服务时，它可以通过传递 ServiceReference 对象到 BundleContext.getSercice(ServiceReference)来获得。请看“服务定位”节。

一个 ServiceReference 对象封装了关于服务对象代表的属性和其它元数据。这个元数据信息可以被一个 Bundle 查询来帮助选择一个最适合他们需求的一个服务。

当一个 Bundle 使用框架服务注册表来查询服务，Framework 必须为请求的 Bundle 提供请求服务的 ServiceReference 对象，而不是服务本身。

一个 ServiceReference 对象只要服务对象注册就有效。然而，它的属性必须保持可用，只要 ServiceReference 对象存在。

5.2.2 服务接口

一个服务接口是服务公共方法的规范。

实际上，一个 Bundle 开发人员通过实现它的服务接口创建服务对象且使用框架服务注册表注册这个服务。一旦 Bundle 以一个接口名称注册一个服务对象，相关的服务可以通过制定接口名称来查询，而且它的方法可以用服务接口的方式来访问。框架也支持使用一个类名注册服务对象，因此在本规范中对服务接口的引用也可以解释为对一个接口或类。

当需要一个框架的服务对象时，一个 Bundle 可以指定服务接口的名称，服务对象必须事先这个接口。在该请求，Bundle 可能也会指定过滤字符串来进一步查询。

很多服务接口由一些组织，比如 OSGi 联盟，来定义和指定。一个服务接口可以作为标准化广泛接受，这个接口可以由任何 Bundle 开发人员实现和使用。

5.2.3 注册服务

一个 Bundle 通过使用框架服务注册表注册一个服务对象来发布一个服务。一个使用框架注册的服务对象暴露给安装在 OSGi 环境的其它 Bundle。

每一个注册的服务对象有一个独一无二的 ServiceRegistration 对象，且有一个或多个引用它的 ServiceReference 对象。这些 ServiceReference 对象暴露了服务对象的注册属性，包括他们实现的服务接口集合。ServiceReference 对象其后可以用于查询一个实现希望的服务接口的服务对象。

Framework 允许 Bundle 来动态的注册和卸载服务对象。因此，一个 Bundle 在 STARTING、ACTIVE 或 STOPPING 装载的任何时候都允许来注册服务对象。

一个 Bundle 通过调用其上下文对象一个 BundleContext.registerService 方法使用框架注册一个服务对象：

●registerService(String,Object,Dictionary)——使用单一的服务接口的服务对象。

●registerService(String[],Object,Dictionary)——使用多个服务接口的服务对象。

一个 Bundle 想要注册的服务对象的服务接口的名称（一个或多个）作为参数提供给 registerService 方法。框架必须确保服务对象确实是指定接口的一个实例，除非这个对象是一个服务工厂。

为了确保这个检查，框架必须为每一个来自一个 Bundle 或一个共享包指定的服务接口加载 Class 对象。对于每一个 Class 对象，Class.isInstance 必须被调用且使用服务对象作为参数时 Class 对象必须返回为 true。

正在被注册的服务对象可能需要通过一个 Dictionary 进一步描述，它作为一个键/值对集合包含了服务的属性。

一个已经被成功注册的服务对象的服务接口名称会作为 objectClass 的键自动添加到服务对象的属性。这个值必须被框架自动设置且由这个 Bundle 提供的任何值必须被覆盖。

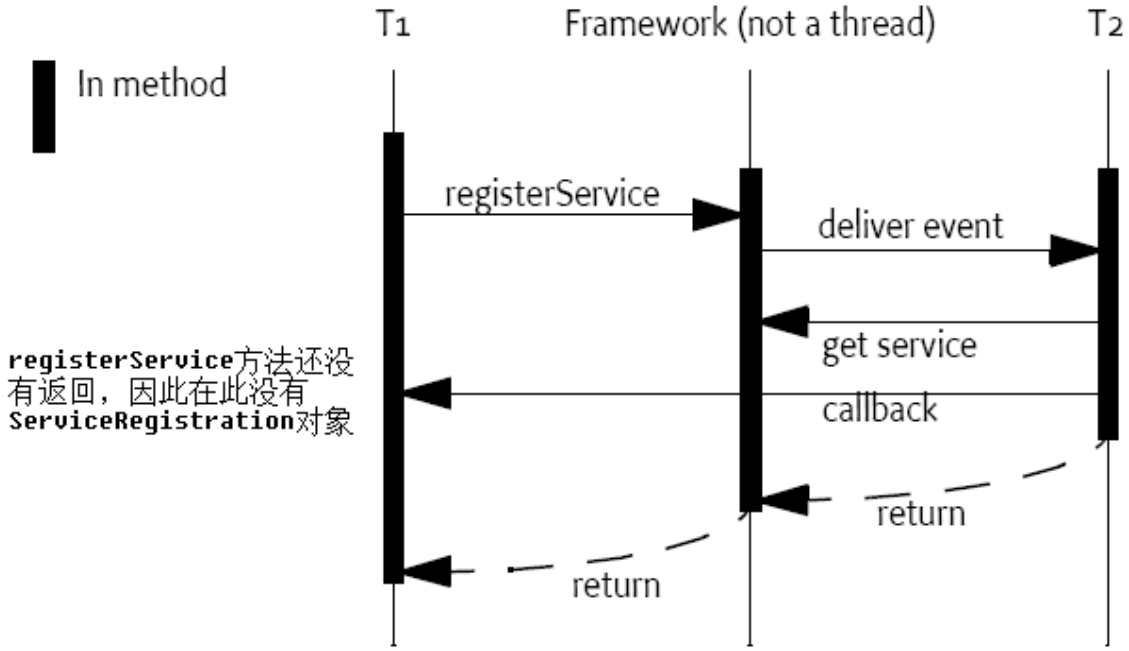
如果服务对象被成功注册，框架必须返回一个 ServiceRegistration 对象给调用者。一个服务对象只能由保留 ServiceRegistration 对象的对象卸载（查看 unregister()方法）。每一个成功的服务对象注册必须也有一个独一无二的 ServiceRegistration 即使多次注册同一个服务对象。

在服务注册之后，使用 ServiceRegistration 对象是更改服务对象的属性的唯一可靠方式（查看 serProperties(Dictionary)）。在服务注册之后，更改一个服务对象的 Dictionary 对象可能不会影响服务的属性。

注册一个服务对象的过程必须受权限检查支配。正在注册的 Bundle 必须有 ServicePermission[<interface name>, REGISTER]来使用所有指定服务接口注册服务对象 。否则，服务对象必须不能被注册且必须抛出一个 SecurityException。

5.2.4 ServiceRegistration 对象的早期需求

一个服务对象的注册将引起所有的 ServiceListener 对象接收通知。这是一个异步通知。这意味着一个监听器可以访问服务并在 registerService 方法返回 ServiceRegistration 对象之前调用服务的方法。在某些情况，ServiceRegistration 对象的访问需要一个回调。然而，正在注册的 Bundle 还没有接收到一个 ServiceRegistration 对象。下图显示了这个顺序。



在上述的情况，对注册对象的访问可以通过一个 ServiceFactory 对象。如果一个 ServiceFactory 对象被注册了，框架必须使用这个 ServiceFactory 的 getService(Bundle,ServiceRegistration)回调正在注册的 Bundle。这要求 ServiceRegistration 对象作为参数传给这个方法。

5.2.5 服务属性

属性作为键/值对集合保存信息。键必须是一个 String 对象且值应该是被过滤对象认识的类型。同一个键的多个值通过使用数组和 Collection 对象支持。

属性的值应该仅限于原始或 Java 标准类型，以阻止不必要的 Bundle 依赖。框架不能检测到通过载 Bundle 通过服务属性交换对象创建的依赖。

一个属性的键是大小写无关的。ObjectClass、OBJECTCLASS 和 objectclass 是相同的属性键。一个框架必须在 ServiceReference.getPropertyKeys 返回以它最后一个设置的键。当传递一个包含仅是大小写区别键的 Dictionary 对象时，框架必须引发一个异常。

服务属性用于来提供服务对象的信息。这些属性不应该用来参与服务的实际功能。更改服务注册的属性是一个潜在的昂贵操作。比如，一个框架可能在注册时候预先将属性处理成索引格式以加快后期查询。

Filter 接口支持复杂的过滤，它可以用来查找匹配的服务对象。因此，在框架服务注册表，所有的属性共享一个名称空间。因此，使用简洁的名称或更短的名称的正规的定义来阻止冲突是很重要的。比一些 OSGi 规范保留了这名名称空间的备件。所有使用 service.前缀开头且 objectClass 属性被保留用于 OSGi 规范。

属性值	类型	常数	属性描述
objectClass	String[]	OBJECTCLASS	objectClass 属性包含了服务对象在框架注册的接口。框架必须自动设置这个属性。框架必须保证党一个服务对象使用 BundleContext.getService(ServiceReference)取出时，它能够自动的转换到任何的接口名称。
service. descripgon	String	SERVICE_ DESCRIPTION	这个属性用于描述且是可选的。框架和 Bundle 可以使用这个属性来提供一个注册的服务对象的简短描述。这个目的主要适用于调试因为它没有支持本地化。
service.id	Long	SERVICE_ID	每一个注册的服务对象都被框架赋予一个独一无二的 service.id。这个数字加到服务对象的属性。框架赋予一个唯一的值到每一个注册的对象，它大于所有提供给以前的服务对象。

service.pid	String	SERVICE_PID	这个属性非强制地为服务对象标识一个持久的、唯一标识。查看“持久标识”节。
Service.ranking	Integer	SERVICE_RANKING	当注册一个服务对象，一个 Bundle 可能会选择的指定这个数字作为服务的一个属性。如果存在多个服务接口，使用最大的 SERVICE_RANKING 数值的服务，或当等于最小的 SERVICE_ID 时，决定了框架返回哪一个服务对象。
Service.vendor	String	SERVICE_VENDER	可选的属性，由 Bundle 使用注册指定了生产商的服务对象。

5.2.6 持久标识（PID）

一个持久标识的目的是来跨框架重启标识服务。服务可能在每次被注册时引用相同的底层实体，因此需要使用一个包含一个 PID 的属性。PID 是一个服务的唯一标识，通过框架的多次调用来持久化服务。如果 Bundle 停止然后又启动，必须使用相同的 PID。

PID 的格式必须是：

pid ::= symbolic-name

一个 PID 必须是每一个服务的唯一标识。一个 Bundle 一定不能为多个服务使用相同的 PID，其它 Bundle 也不能使用相同的 PID。如果出现这样的情况，那就是一个错误条件。

5.2.7 定位服务

为了使用一个服务对象且调用它的方法，一个 Bundle 必须首先获得一个 ServiceReference 对象。BundleContext 接口定义了两个方法，一个 Bundle 可以调用来从框架获取 ServiceReference 对象：

●getServiceReference(String)——这个方法返回一个 ServiceReference 对象，它对应了实现的服务对象，且使用指定的服务接口注册。如果存在多个对象，这个服务对象是以最高的 SERVICE_RANKING 返回的对象。

●getServiceReference(String,String)——这个方法返回 ServieReference 对象的数组：

A) 实现和使用给定的接口注册。

B) 满足指定的搜索过滤。

如果没有匹配的服务对象返回，这两个方法都必须返回 null。否则，调用者接收一个或多个 ServiceReference 对象。这些对象可以用于取出底层服务对象的属性，或者用于通过 BundleContext 对象获取实际的服务对象。

这两个方法都要求调用者有 ServicePermission[<name>,GET]来获取指定接口名称的服务对象。如果调用者没有要求的权限，这些方法必须返回 null。

5.2.8 获取服务属性

为了允许服务对象的互操作，ServiceReference 接口定义了这两个方法：

●getPropertyKeys()——返回可用的键的数组。

●getProperty(String)——返回一个属性的值。

这两个方法都必须继续提供关于引用的服务对象的信息，即使它已经从框架卸载了。当一个 ServiceReference 对象使用日志服务存储时，这个需求是有用。

5.2.9 获取服务对象

BundleContext 对象用于获取实际的服务对象，这样框架可以管理依赖型。如果一个 Bundle 取出一个服务对象，那个 Bundle 在注册的服务对象的生命周期中将开始依赖了。这个依赖由用于获取服务对象的 BundleContext 对象来跟踪，且是当与其它 Bundle 共享 BundleContext 共享需要很小心的一个原因。

方法 BundleContext.getService(ServiceReference)返回一个实现在 objectClass 属性定义的接口的对象。

这个方法具有以下的典型特点：

●如果底层服务对象已经被卸载，则返回为 null。

●如果调用者有 ServicePermission[<interface name>,GET]，它使用服务注册的至少一个接口来获取服务对象。这个权限检测是必须的，这样 ServiceReference 可以无需过多安全限制自由传递。

●为该 BundleContext 增加服务一次对象使用计数器。

●如果服务对象没有实现 ServiceFactory 接口，它直接被返回。否则，Bundle 上下文服务对象使用计数器为 1，这个对象给转换成 ServiceFactory 对象且调用 getService 方法为调用的 Bundle 来创建一个自定义的服务对象并然后返回。要不然，返回这个自定义对象的缓存备份。

5.2.10 关于服务的信息

Bundle 接口定义了两个方法来返回关于 Bundle 的服务使用的信息：

●getRegisteredServices()——返回 Bundle 已经注册到框架的服务对象。

●getServicesInUser()——返回 Bundle 正在使用的服务对象。

5.3 服务事件

●ServiceEvent——为服务对象报告注册、卸载和属性更改。这类的所有事件都必须被异步传输。事件的类型由 getType()方法给定，它返回一个 int。时间类型在将来可以被扩展，不认识的事件类型必须被忽略。

●ServiceListener——当一个服务对象被注册，或更改，或者正在卸载是，使用一个 ServiceEvent 调用。当一个 ServiceEvent 发生时，必须为每一个注册的监听器执行安全检查。监听器必须不能被调用除非注册监听器的 Bundle 已经为服务对象注册的至少一个接口请求 ServicePermission[<interface name>,GET]。

使用一个服务对象的 Bundle 应该注册一个 ServiceListener 对象来跟踪这个服务对象的可用性，且当服务被卸载后需要采取合适的操作。

5.4 旧的引用

框架必须管理 Bundle 间的依赖。这种管理然而被限制在框架结构里。Bundle 必须监听由框架产生的事件来清理和删除旧的引用。

一个旧的引用是一个属于已经停止的 Bundle 的类加载器的 Java 对象或是一个关联已经卸载的服务对象。标准 Java 没有提供任何通用的方法来清理旧引用，且 Bundle 开发人员必须仔细的分析代码来确保删除旧引用。

旧引用具有潜在的害处，因为他们阻碍了 Java 垃圾收集来获取这些停止的 Bundle 的类和也可能是实例。这将明显增加内存使用且可能引起更新本地代码库失败。使用服务的 Bundle，强烈建议使用服务跟踪器或者声明式服务。

服务开发人员可以通过以下机制来最小化旧引用数量（但不能阻止）：

●使用 ServiceFactory 接口实现服务对象。ServiceFactory 接口的方法简化使用服务对象的 Bundle 的跟踪。查看“服务工厂”节。

●间接的使用服务对象实现。由其它 Bundle 获得的服务对象应该使用一个指针来指向真正的服务对象实现。当服务对象不可用时，这个指针变为 null，能够有效的减少对实际服务对象的引用。

一个卸载的服务的行为是不确定的。这样的服务可能会继续工作，或者会根据条件抛出异常。这种类型的错误必须记录日志。

5.5 过滤

框架必须提供一个 Filter 接口，并且使用在 getServiceReference 方法使用的过滤器语法，这个语法在“过滤器语法”节定义。Filter 对象可以通过调用使用过滤字符串的 BundleContext.createFilter(String)或 FrameworkUtil.createFilter(String)方法来创建过滤器。过滤器支持以下匹配方法：

●match(ServiceReference)——以忽略大小写方式查询键来匹配服务引用的属性。

●match(Dictonary)——以忽略大小写方式查询键来匹配给定词典的项目。

●matchCase(Dictonary)——以区分大小写方式查询键来匹配给定词典的项目。

一个过滤器对象可以使用无数次来决定匹配参数——一个 ServiceReference 对象或一个 Dictionary 对象，是否匹配用于创建过滤器的过滤字符串。

这个匹配需要比较过滤器的字符串的值和来自服务属性或词典的对象。这种比较可以使用 Comparable 接口来执行，如果目标对象的类实现了一个使用单一字符串的构造器且类实现了 Comparable 接口。也就是说，如果目标对象是类 Target 的实例，类 Target 必须实现：

●一个构造器 Target(String)。

●实现 java.lang.Comparable 接口。

如果目标对象没有实现 java.lang.Comparable 接口，当对象相等（使用 equals(Object)方法）时，=、～=、<=和>=操作符必须返回 true。类 Target 不一定需要是一个公共类。

以下例子显示了一个类如何使用过滤器校验枚举的次序。

```
public class B implements Comparable {
        String keys[] = {"bugs", "daffy", "elmer", "pepe"};
        int index;
        public B(String s) {
                for ( index=0; index<keys.length; index++ )
                if ( keys[index].equals(s) )
                        return;
        }
        public int compareTo( Object other ) {
                B vother = (B) other;
                return index - vother.index;
        }
}
```

这个类可以使用以下过滤器来使用：

(!(enum >= elmer)) -> 匹配“bugs”和“daffy”。

方法 Filter.toString 必须总是返回过滤字符串，但无需删除空格。

5.6 服务工厂

一个服务工厂允许自定义当一个 Bundle 调用 BundleContext. getService(ServiceReference)返回的服务对象。

通常的，由一个 Bundle 注册的服务对象直接返回。然而，如果注册的服务对象实现了 ServiceFactory 接口，框架必须调用这个对象的方法来为每一个需要获取这个服务的不同 Bundle 创建一个独一无二的服务对象。

当服务对象不再被一个 Bundle 使用，比如，当一个 Bundle 停止了，那么框架必须通知 ServiceFactory 对象。

ServiceFactory 对象帮助管理那些没有被框架显示管理的 Bundle 依赖。通过绑定一个服务对象到请求的 Bundle，这个服务在 Bundle 停止使用这个服务时能够被通知，比如当它停止了，且释放了提供给那个 Bundle 的服务的资源。

ServiceFactory 接口定义了以下方法：

●getService(Bundle,ServiceRegistration)——如果调用了 BundleContext. getService 且以下为 true 情况下，这个方法由框架调用。

A) BundleContext.getService 的 ServiceReference 参数指向了实现 ServiceFactory 接口的服务对象。

B) 服务对象的 Bundle 使用计数器为 0，也就是说，这个 Bundle 没有对这个服务对象的任何依赖。

对 BundleContext.getService 的调用必须由框架路由到这个方法，并将调用者的 Bundle 传送给它。框架必须缓存请求 Bundle 到服务的映射，而且在将来只要请求的 Bundle 的服务使用计数器大于 0，对 BundleContext.getService 的调用会返回缓存的服务对象。

框架必须检查由这个方法返回的服务对象。如果这不是当一个注册的服务工厂的所有类名的一个实例时，必须返回 null 给调用 getService 的调用者。这种检查必须在“注册服务”节执行。

●ungetService(Bundle,ServiceRegistration,Object)——如果调用了 BundleContext.ungetService 且以下条件为真这个方法将被框架调用：

A) BundleContext.ungetService 的 ServiceReference 参数指向了实现 ServiceFactory 接口的服务对象。

B) Bundle 在这个调用返回后，对这个服务对象的使用计数器必须设置为 0，也就是说，Bundle 即将释放对这个服务对象的最后依赖。

对 BundleContext.ungetService 的调用必须由框架路由到这个方法，这样 ServiceFactory 对象可以释放先前创建的对象。

此外，先前创建服务对象的缓存拷贝必须由框架解除引用，这样它可以被垃圾收集。

5.7 释放服务

为了一个 Bundle 来释放一个服务，它必须删除对注册服务对象的 Bundle 的动态依赖。BundleContext 接口定义了一个方法来释放服务：ungetService (ServiceReference)。一个服务引用对象作为参数传递到这个方法。

这个方法返回一个布尔值：

●如果这个服务对象的 Bundle 使用计数器在调用这个方法时已经为 0，或服务对象已经被卸载，则返回 false。

●如果调用这个方法之前服务对象的 Bundle 使用计数器大于 0，返回 true。

5.8 卸载服务

ServiceRegistration 接口定义了 unregister()方法来卸载一个服务对象。这必须从框架服务注册表中删除服务对象。这个 ServiceRegistration 对象对应的 ServiceReference 对象再也不能用于访问服务对象了。

事实上，这个方法在 ServiceRegistration 对象上，将确保只有存放这个对象的 Bundle 可以卸载关联的服务对象。然而，卸载一个服务对象的 Bundle，可能不一定是注册它的 Bundle。比如，注册的 Bundle 可以传递一个 ServiceRegistration 对象到另一个 Bundle，使那个 Bundle 具有卸载服务对象的职责。

当 ServiceRegistration.unregister 成功完成，这个服务对象必须：

●从框架服务注册表彻底删除。因此获取那个服务对象的 ServiceReference 再也不能用于访问服务对象了。使用给定的 ServiceReference 对象来调用 BundleContext.getService 方法必须返回 null。

●必须被卸载，即使其它 Bundle 已经依赖它。必须通过发布一个 ServiceEvent.UNREGISTERING 类型的 ServiceEvent 对象来通知 Bundle 服务已经被卸载。这个事件异步发送，以让给定的 Bundle 有机会释放服务对象。

在接收到一个 ServiceEvent.UNREGISTERING 类型的事件，一个 Bundle 应用释放服务对象和释放对这个对象的任何引用，这样服务对象变可以被 JAVA 虚拟机的垃圾收集回收。

●由所有使用的 Bundle 释放。在所用 ServiceListener 对象的调用返回之后，对于每一个对这个服务对象的使用计数器大于 0 的 Bundle，框架必须设置计数器为 0 并释放这个服务对象。

5.9 多版本导出考虑

允许多个版本使用给定的名称导出一个包会增加框架实现者和 Bundle 编程人员的复杂性：类名不再唯一标识导出的类。这影响了服务注册和权限检查。

5.9.1 服务注册

Bundle 必须不能暴露出与 Bundle 类加载器冲突的服务。一个提供一个服务的 Bundle 应该能够被期望它可以安全的将服务对象转换成服务关联的接口或类且可以被访问。不应该发生任何 ClassCastExceptions，因为这些接口没有来自同一个类加载器。服务注册表因此必须保证 Bundle 只能看到与其兼容的服务。一个服务不与一个正在获取而不是注册服务的 Bundle 兼容，当那个 Bundle 没有连线到接口包的源类加载器，也就是说，要么它连线到相同的源类加载器，或者它本根没有连线到任何包。

Bundle 不能偶合的接触到不兼容服务是首要的。因此，以下方法需要根据调用 Bundle 与接口的不兼容来过滤掉 ServiceReference 对象。Bundle 通过 BundleContext 来标识：

●getServiceReference(String)——对于指定接口，只有返回一个与调用 Bundle 兼容的 ServiceReference。

●getServiceReferences(String,String)——对于指定的接口，只有返回与调用 Bundle 兼容的那些 ServiceReference 对象。

方法 getAllServiceReferences(String,String)提供了不使用兼容限制访问服务注册表。通过这个方法查询的服务可能引起类型转换异常来纠正类名称。

ServiceReference.isAssignableTo(Bundle,String)方法也可以用于测试通过 ServiceReference 注册引用的 Bundle 和指定 Bundle 是否都连线到指定接口相同来源。

5.9.2 服务事件

服务事件必须只能传输到与服务引用兼容的事件监听器。

一些 Bundle 需要监听所有的服务事件，而不关心兼容性问题。因此添加了一个新的服务监听器：AllServiceListener。这个一个标记接口，它扩展 ServiceListener。使用这个标记接口的监听器向框架表明它们想要看到所有的服务，包括与他们不兼容的服务。

5.10 安全性

5.10.1 服务权限

一个服务权限有以下参数：

●接口名称（Interface Name）——接口名称可能以通配符结尾用于匹配多个接口名称。（查看 java.security.BasicPermisson 关于通配符的讨论）。

●操作（Action）——支持的操作有：

A) REGISTER——指定权限拥有者可以注册服务对象。

B) GET——指定权限拥有者可以获取服务。

当一个对象使用 BundleContext.registerService 被注册为一个服务对象，这个注册的 Bundle 必须拥有 ServicePermission 来注册所有指定名称的类。查看“注册服务”节。

当一个 ServiceReference 对象使用 BundleContext.getServiceReference 或者 BundleContext.getServiceReferences 从服务注册表获取，调用 Bundle 必须请求 ServicePermission[<interface name>, GET]来使用指定的类名获取服务对象。查看“服务引用”节。

当使用 BundleContext.getService(ServiceReference)从一个 ServiceReference 对象获取一个服务对象，调用代码必须为服务注册的至少一个类请求 ServicePermission[<name>,GET]来获取服务对象。

ServicePermission 对通过服务监听器接受的服务事件必须作为一个过滤器使用，也包括枚举服务的那些方法，包括 Bundle.getRegisteredServices 和 Bundle.getServicesInUser。框架必须确保如果一个 Bundle 没有访问权限，它不可以检测到一个服务是否存在。

5.11 变更（略）

6 框架 API

6.1 org.osgi.framework

框架包版本 1.4。

需要使用这个包的 Bundel 必须在 Bundle 清单的 Import-Package 头信息列出这个包。比如：

Import-Package: org.osgi.framework; version=1.4

6.1.1 概要

●AdminPermission——一个对执行指定需要特权的管理操作或获取关于 Bundle 的敏感信息的 Bundle 授权。

●AllServiceListener——一个没有基于包连接线过滤的 ServiceListener 监听器。

●Bundle——框架中一个安装的 Bundle。

●BundleActivator——自定义 Bundle 的启动和停止。

●BundleContext——在框架中一个 Bundle 的执行上下文。

●BundleEvent——一个来自框架的事件，它描述一个 Bundle 生命周期变更。

●BundleException——一个框架异常，用于表明发生了一个 Bundle 生命周期异常。

●BundleListener——一个 BundleEvent 的监听器。

●BundlePermission——一个对请求或提供一个 Bundle 或附加片段的 Bundle 授权。

●Configurable——支持一个配置对象。

●Constants——定义了 OSGi 环境系统属性、服务属性和清单头信息属性键的标准名称。

●Filter——一个基于 RFC1960 的过滤器。

●FrameworkEvent——一个来自框架的通用事件。

●FrameworkListener——一个 FrameworkEvent 监听器。

●FrameworkUtil——框架辅助类。

●InvalidSyntaxException——一个用于表明过滤字符串有一个无效语法的框架异常。

●PackagePermission——一个对引用或导出包的 Bundle 授权。

●ServiceEvent——一个来自框架的事件，用于给描述服务声明周期变更。

●ServiceFactory——允许服务在 OSGi 环境中来提供自定义的服务对象。

●ServiceListener——一个 ServiceEvent 监听器。

●ServicePermission——一个对注册或服务服务的 Bundle 授权。

●ServiceReference——对一个服务的引用。

●ServiceRegistration——一个注册的服务。

●SynchronousBundleListener——一个同步 BundleEvent 监听器。

●Version——Bundle 和包的版本标识。

6.1.2 public final class AdminPermission extends BasicPermission

一个对执行指定需要特权的管理操作或获取关于 Bundle 的敏感信息的 Bundle 授权。这个权限的操作有：

操作	方法
class	Bundle.loadClass
execute	Bundle.start
extensionLifecycle	为扩展 Bundle 的 BundleContext.installBundle 为扩展 Bundle 的 Bundle.update 为扩展 Bundle 的 Bundle.uninstall
lifecycle	BundleContext.installBundle Bundle.update Bundle.uninstall
listener	为 SynchronousBundleListener 的 BundleContext. addBundleListener 和 removeBundleListener
metadata	Bundle.getHeaders Bundle.getLocation
resolve	PackageAdmin.refreshPackage PackageAdmin.resolveBundles
resource	Bundle.getResource Bundle.getResources Bundle.getEntry Bundle.getEntryPaths Bundle.findEntries Bundle resource/entry URL 创建
startlevel	StartLevel.setStartLevel StartLevel.setInitialBundleStartLevel
context	Bundle.getBundleContext

特殊操作“*”表示所有操作。

这个权限的命名是一个过滤表达式。这个过滤表达式能访问以下参数：

●数字签名——用于签名一个 Bundle 的可区分的名字链（Distinguished Name chain）。在一个可区分名字的通配符不能根据过滤规则来匹配，但可以根据 DN 链定义的规则。

●位置——Bundle 的位置。

●ID——指定 Bundle 的 Bundle ID。

●名称——一个 Bundle 的特征名称。

6.1.2.1 public static final String CLASS = “class”

“class”操作字符串（值为 class）。

6.1.2.2 public static final String CONTEXT = “context”

“context”操作字符串（值为 context）。

6.1.2.3 public static final String EXECUTE = “execute”

“execute”操作字符串（值为 execute）。

6.1.2.4 public static final String EXTENSIONLIFECYCLE = “extensionLifecycle”

“extensionLifecycle”操作字符串（值为 extensionLifecycle）。

6.1.2.5 public static final String LIFECYCLE = “lifecycle”

“lifecycle”操作字符串（值为 lifecycle）。

6.1.2.6 public static final String LISTENER = “listener”

“listener”操作字符串（值为 listener）。

6.1.2.7 public static final String METADATA = “metadata”

“metadata”操作字符串（值为 metadata）。

6.1.2.8 public static final String RESOLVE = “resolve”

“resolve”操作字符串（值为 resolve）。

6.1.2.9 public static final String RESOURCE = “resource”

“resource”操作字符串（值为 resource）。

6.1.2.10 public static final String STARTLEVEL = “startlevel”

“startlevel”操作字符串（值为 startlevel）。

6.1.2.11 public AdminPermission()

创建一个新的 AdminPermission 对象，用于匹配所有的 Bundle 且有所有的操作。等价于 AdminPermission(“*”,”*”)。

6.1.2.12 public AdminPermission(String filter, String actions)

filter——一个过滤表达式，可以使用签名、位置、ID 和名称键。“*”或 null 将匹配所有的 Bundle。

actions——以上所有操作。“*”或 null 表示所有操作。

用于创建一个 AdminPermission。这个构造器必须仅用于创建一个即将被检测的权限。

例如：

(signer = *, o = ACME, c = US)

(&(signer = *, o = ACME, c=US) (name = com.acme.*) (location = http://www.acme.com/bundles/*))

(id >= 1)

当在过滤表达式中使用一个签名键，这个签名的值必须忽略特殊过滤字符（’*’, ‘(’, ‘)’）。

null 参数等价于“*”。

6.1.2.13 public AdminPermission(Bundle bundle, String actions)

Bundle——一个 Bundle。

actions——同上。

创建一个新的 AdminPermission 对象，用于需要检测一个 Permission 对象的代码。

6.1.2.14 public boolean equals(Object obj)

obj——用于比较与这个对象是否相等的对象。

决定两个 AdminPermission 对象是否相等。

6.1.2.15 public String getActions()

返回一个 AdminPermission 所有操作的正规字符串表达式。

总是以以下次序返回现有的 AdminPermission 操作：class, execute, extensionLifecycle, lifecycle, listener, metadata, resolve, resource, startlevel, context。

返回值——AdminPermission 操作的正规字符串表达式。

6.1.2.16 public int hashCode()

返回这个对象的 hash 码。

返回值——这个对象的 hash 码。

6.1.2.17 public boolean implies(Permission p)

p——查询的权限。

决定是否指定的权限由这个对象隐含。如果指定的权限并没有使用 Bundle 构建，这个方法将抛出一个异常。

这个方法将返回 true，如果指定的权限是一个 AdminPermission

并且（1）

●这个对象过滤器与指定权限的 Bundle ID、Bundle 位置和 Bundle 数字签名者区分名称链匹配，或者

●这个对象的过滤表达式是“*”。

并且（2）

这个对象的操作包含指定权限所有操作。

特殊情况：如果指定权限使用“*”过滤器构建，那么如果这个对象的过滤器是“*”且这个对象操作包含指定权限所有操作时这个方法返回 true。

返回值——如果指定权限由这个对象隐含返回 true，否则返回 false。

抛出异常——如果指定权限没有使用一个 Bundle 或“*”构建则抛出 RuntimeException。

6.1.2.18 public PermissionCollection newPermissionCollection()

返回一个适合存储的 AdminPermissions 的新 PermissionCollection 对象。

返回值——一个新的 PermissionCollection 对象。

6.1.3 public interface AllServiceListener extends ServiceListener

一个没有基于包连接线过滤的 ServiceEvent 监听器。

AllServiceListener 是一个监听器接口，由 Bundle 开发人员实现。当触发一个 ServiceEvent，它将被同步的传输到一个 AllServiceListener。框架可能不按顺序传递一些 ServiceEvent 对象到一个 AllServiceListener 并且可能并发的调用且/或重新进入一个 AllServiceListener。

一个 AllServiceListener 对象通过使用 BundleContext.addServiceListener 方法注册在框架中。AllServiceListener 对象由一个 ServiceEvent 对象在服务注册、更改或卸载过程调用。

传输到 AllServiceListener 对象的 ServiceEvent 对象通过在注册监听器时指定的过滤器过滤。如果 JRE 支持权限，那么将执行额外的过滤。只有当定义这个监听器的 Bundle 有使用服务注册时至少一个类名称获取这个服务的合适权限，ServiceEvent 对象才能传输到一个监听器。

不想普通 ServiceListener 对象，AllServiceListener 对象不需要考虑监听 Bundle 的源包是否等于注册服务的 Bundle 的源包，而是直接接收所有 ServiceEvent 对象。这意味着如果取出服务对象，监听器可能不能将服务对象转换成合适的服务接口。

查看：ServiceEvent,ServicePermission

起始版本：1.3

并行性：线程安全

6.1.4 public interface Bundle

表示 Framework 中一个安装的 Bundle。

一个 Bundle 对象是一个安装的 Bundle 的生命周期的访问入口。每一个安装在 OSGi 环境的 Bundle 必须有一个关联的 Bundle 对象。

一个 Bundle 必须有一个唯一的标识，一个由 Framework 生成的长整型。这个标识在 Bundle 的生命周期中必须是不能改变的，即使 Bundle 被更新了。卸载然后重新安装 Bundle 必须创建一个新的唯一的表示。

一个 Bundle 可以是以下六个状态之一：

●UNINSTALLED。

●INSTALLED。

●RESOLVED。

●STARTING。

●STOPPING。

●ACTIVE。

这些状态的值没有规定的顺序；他们使用位值，这些位值可以用于决定一个 Bundle 是否处在一个有效的状态。

一个 Bundle 必须只有当它的状态是 STARTING、ACTIVE 或 STOPPING 时，它才可以执行代码。一个卸载的 Bundle 不能够更改状态了。

Framework 是为一个允许创建 Bundle 对象的实体，且这些对象只有在创建它们的 Framework 中有效。

6.1.4.1 public static final int ACTIVE = 32

这个 Bundle 现在正在运行。只有当一个 Bundle 被成功启动且激活后，它处在 ACTIVE 状态。

6.1.4.2 public static final int INSTALLED = 2

这个 Bundle 已经安装但还没有被解析。当一个 Bundle 被 Framework 安装但没有货无法被解析时，它处于 INSTALLED 状态。

如果 Bundle 的代码依赖没有被解析，这个状态出现。Framework 可能会尝试解析一个 INSTALLED 状态的 Bundle 的代码依赖然后将其状态迁移到 RESOLVED 状态。

6.1.4.3 public static final int RESOLVED = 4

Bundle 已经被解析而且可以被启动了。

当 Framework 成功解析了 Bundle 的代码依赖后，一个 Bundle 处在 RESOLVED 状态。这些依赖包括：

●来自 Constants.BUNDLE_CLASSPATH 清单头信息的 Bundle 类路径。

●来自 Constants.EXPORT_PACKAGE 和 Constants.IMPORT_PACKAGE 清单头信息的 Bundle 包依赖。

●来自 Constants.REQUIRE_BUNDLE 清单头信息的需要的 Bundle 依赖。

●来自 Constants.FRAGEMENT_HOST 清单头信息的片段 Bundle 的宿主依赖。

需要注意的是，此时 Bundle 还没有激活。一个 Bundle 在启动之前必须处于 RESOLVED 状态。Framework 可能在某一时刻尝试解析一个 Bundle。

6.1.4.4 public static final int START_ACTIVATION_POLICY = 2

Bundle 启动操作必须根据 Bundle 声明的激活策略激活 Bundle。

这个标志可能在调用 start(int)方法时设置来通知 Framework，这个 Bundle 必须使用 Bundle 声明的激活策略来激活。

6.1.4.5 public static final int START_TRANSIENT= 1

Bundle 的启动操作时临时的并且该 Bundle 的持久化的 autostart 设置不会被改变。

这个标志可能在调用 start(int)方法时设置用于通知 Framework 这个 Bundle 的 autostart 设置不会被更改。如果这个标志没有被设置，那么这个 Bundle 的 autostart 设置将被设置。

6.1.4.6 public static final int STARTING = 8

当前 Bundle 正在启动的过程。

当 Bundle 的 start 方法被调用后，一个 Bundle 处于 STARTING 状态。当 BundleActivator.start 方法被调用后，一个 Bundle 必须处于该状态。如果 BundleActivator.start 方法没有任何异常的完成，那么意味着 Bundle 已经成功启动并必须移动到 ACTIVE 状态。

如果 Bundle 有晚激活策略，那么 Bundle 可能维持在这个状态直到激活被触发。

6.1.4.7 public static final int STOP_TRANSIENT = 1

Bundle 停止是瞬时的，其 autostart 持久化设置不会被更改。

这个标志位当 stop(int)调用时设置，用于通知 Framework 这个 Bundle 的 autostart 设置不会被更改。如果这个标志位没有被设置，那么 Bundle 的 autostart 设置会被更改。

6.1.4.8 public static final int STOPPING = 16

这个 Bundle 正在停止。

当 Bundle 的 stop 方法被调用后，Bundle 处于正在停止状态。一个 Bundle 当 BundleActivator.stop 方法被调用时，必须处于这个状态。当 BundleActivator.stop 方法成功执行后，它必须迁移到 RESOLVED 状态。

6.1.4.9 public static final int UNINSTALLED = 1

这个 Bundle 已经被卸载而且不可用了。

这个 UNINSTALLED 状态只有在 Bundle 被卸载后可视，这个 Bundle 处于一个不可用的状态但是对 Bundle 对象的引用还是可用且可以用于查看属性。

6.1.4.11 public BundleContext getBundleContext()

（1）说明

返回这个 Bundle 的上下文。返回的 BundleContext 可以被调用者使用来操作这个 Bundle 的行为。

如果这个 Bundle 不是在 STARTING，ACTIVE 或 STOPPING 状态，或者这个 Bundle 是一个片段 Bundle，那么这个 Bundle 没有一个可用的 BundleContext。如果这个 Bundle 没有一个有效的 BundleContext，那么这个方法将返回 null。

（2）异常

SecurityException——如果调用者没有合适的 AdminPermission[this,CONTEXT]且 JRE 支持权限时，抛出这个异常。

6.1.4.12 public long getBundleId()

返回这个 Bundle 的唯一标识。这个 Bundle 在被 OSGi 环境安装时赋予了一个唯一的标识。

一个唯一的标识有以下属性：

●是唯一的且持久化的。

●是一个长整型。

●他的值不能被另一个 Bundle 使用，即使一个 Bundle 已经被卸载了。

●当一个 Bundle 维持在被安装的状态时，不能改变。

●当一个 Bundle 被更新时，不能改变。

这个方法在 Bundle 处于 UNINSTALLED 状态时，仍需要返回这个 Bundle 的唯一标识。

6.1.4.13 public URL getEntry(String path)

（1）参数

path：资源的路径名称。

（2）说明

返回一个对应在这个 Bundle 指定路径的资源的 URL。这个 Bundle 的装载器不会用于搜索这个资源。只有这个 Bundle 的内容被搜索。

指定的资源必须是相对于这个 Bundle 的根路径且可能以“/”开始。一个“/”路径值指定了这个 Bundle 的根路径。

（3）返回值

对应这个资源的 URL，如果没有找到对应的项目或没有合适的 AdminPermission[this,RESOURCE]且 JRE 支持权限时返回 null。

（4）异常

IllegalStateException——如果这个 Bundle 已经被卸载。

6.1.4.14 public Enumeration getEntryPaths(String path)

（1）参数

用于返回一些资源路径的名称（如 readme.txt 文件名称）。

（2）说明

返回在这个 Bundle 那些最长子路径的路径与指定路径匹配的资源的 Enumeration 对象。这个 Bundle 的类加载器没有用于搜索这些资源。只有这个 Bundle 的内容被搜索。

指定的资源必须是相对于这个 Bundle 的根路径且可能以“/”开始。一个“/”路径值指定了这个 Bundle 的根路径。

返回以“/”结尾标志子路径的一些路径。返回的所有路径都是相对于这个 Bundle 的根且不能以“/”开始。

（3）返回值

对应这个资源名的所有 URL，如果没有找到对应的项目或没有合适的 AdminPermission[this,RESOURCE]且 JRE 支持权限时返回 null。

（4）异常

IllegalStateException——如果这个 Bundle 已经被卸载。

6.1.4.15 public Dictionary getHeaders()

（1）说明

返回这个清单所有头部和值。这个方法将根据这个 Bundle 的清单文件的主段获取清单所有头部和值，也就是在第一个空行前的所有行。

清单头信息名称是大小写区分的。这个方法返回的 Dictonary 对象必须以大小写敏感的方式处理头信息名称。如果一个清单头信息值以“%”开头，它必须根据默认的区域设置进行本地化。

比如，如果以下清单头信息和值在清单文件中存在，则他们将被包括在内：

Bundle-Name

Bundle-Vendor

Bundle-Version

Bundle-Description

Bundle-DocURL

Bundle-ContactAddress

这个方法必须一直能够返回头信息直到这个 Bundle 处在 UNINSTALLED 状态。

（2）返回值

返回包含这个 Bundle 的清单头部和值的一个 Dictionary 对象。

（3）异常

SecurityException——如果调用者没有合适的 AdminPermission[this,METADATA]且 JRE 支持权限。

6.1.4.16 public Dictionary getHeaders(String locale)

（1）参数

参数 locale：指定头信息值被本地化的区域的名称。如果指定的区域是 null，那么这个 locale 将使用 java.util.Locale.getDefault 的返回值。如果指定的 locale 是一个空字符串，这个方法将返回包含“%”原始的清单头信息。

（2）说明

返回使用指定区域本地化的清单头部和值。

这个方法执行了和 Bundle.getHeaders()相同的功能，除了它使用指定的区域本地化清单头部的值。

如果一个清单头部的值以“%”开头，它必须根据指定的区域进行本地化。如果一个区域被指定但找不到，那么清单头部值必须使用默认的区域返回。本地化搜索按以下顺序：

bn + “_” + Ls + “_” + Cs + “_” + Vs

bn + “_” + Ls + “_” + Cs

bn + “_” + Ls

bn + “_” + Ld + “_” + Cd + “_” + Vd

bn + “_” + Ld + “_” + Cd

bn + “_” + Ld

bn

其中，bn 是这个 Bundle 本地化的基本名称，Ls、Cs 和 Vs 是指定的区域（language——语言，country——国家，variant——变量），Ld、Cd 和 Vd 是默认区域。如果指定的区域是 null，那么这个 locale 将使用 java.util.Locale.getDefault 的返回值。如果指定的 locale 是一个空字符串，这个方法将返回包含“%”原始的清单头信息。

这个方法必须一直能够返回头信息直到这个 Bundle 处在 UNINSTALLED 状态，而且头部值必须是原始或默认区域本地化的值。

（3）返回值

一个包含这个 Bundle 清单头部和值的 Dictionary 对象。

（4）异常

SecurityException——如果调用者没有合适的 AdminPermission[this,METADATA]且 JRE 支持权限。

6.1.4.17 public long getLastModified()

（1）说明

返回这个 Bundle 被最后一次更改的时间。一个 Bundle 只有当它被安装、更新或卸载才被认为是被更改了。这个时间值是从 1970 年 1 月 1 日 00：00：00GMT 开始的毫秒数。

（2）返回值

这个 Bundle 最后一个被更改的时间。

6.1.4.18 public String getLocation()

（1）说明

返回这个 Bundle 位置的标识。

这个 location 标识是在 Bundle 被安装时传递给 BundleContext.installBundle 的位置。这个位置标识在 Bundle 处于安装时不能被更改，即使这个 Bundle 被更新了。

这个方法必须一致返回 Bundle 的位置标识直到这个 Bundle 处于 UNINSTALLED 状态。

（2）返回值

一个表示 Bundle 位置标识的字符串。

（3）异常

SecurityException——如果调用者没有合适的 AdminPermission[this,METADATA]且 JRE 支持权限。

6.1.4.19 public ServiceReference[] getRegisteredServices()

（1）说明

返回这个 Bundle 的注册的所有服务的 ServiceReference 列表，如果没有注册则 null。

如果 JRE 支持权限，一个指定服务的 ServiceReference 对象只有当调用者有 ServicePermission 权限使用注册的服务至少一个声明的类名称来获取服务时才能被包含在这个列表中。

这个列表在调用这个方法的任何时候有效，然而，由于 Framework 是一个非常动态的环境，服务可能在任何时候被更改或卸载。

（2）返回值

一个 ServiceReference 对象数组或 null。

（3）异常

IllegalStateException——如果这个 Bundle 已经被卸载。

6.1.4.20 public URL getResource(String name)

（1）参数

参数 name：资源的名称。查看 java.lang.ClassLoader.getResource 关于资源名称的描述。

（2）说明

查找这个 Bundle 指定的资源。这个 Bundle 的类加载器被调用来搜索指定的资源。如果这个 Bundle 的状态是 INSTALLED，这个方法将在尝试获取资源前尝试去解析这个 Bundle。如果这个 Bundle 不能被解析，那么仅搜索这个 Bundle 来获取指定资源。当这个 Bundle 没有被解析时，引用的包不能被搜索。如果这个 Bundle 是一个片段，则返回 null。

（3）返回值

返回一个指定资源的 URL，或者如果这个资源没有找到、这个 Bundle 是一个片段或调用者没有 AdminPermission[this,RESOURCE]但 JRE 支持权限，则返回 null。

（4）异常

IllegalStateException——如果这个 Bundle 已经被卸载。

6.1.4.21 public Enumeration getResource(String name) throws IOException

（1）参数

参数 name：资源的名称。查看 java.lang.ClassLoader.getResource 关于资源名称的描述。

（2）说明

查找这个 Bundle 指定的资源。这个 Bundle 的类加载器将被调用来搜索指定资源。如果这个 Bundle 的状态是 INSTALLED，这个方法在尝试获取资源前将尝试解析它。如果解析失败，那么仅搜索当前 Bundle 来查找指定资源。如果 Bundle 没有解析，引用包不能搜索。如果 Bundle 是一个片段，那么直接返回 null。

（3）返回值

一个指定名称的资源对应的 URLs Enumeration 对象。如果这个资源没有找到、这个 Bundle 是一个片段或调用者没有 AdminPermission[this,RESOURCE]但 JRE 支持权限，则返回 null。

（4）异常

IllegalStateException——如果这个 Bundle 已经被卸载。

IOException——如果有一个 I/O 错误。

6.1.4.22 public ServiceReference[] getServiceInUser()

（1）说明

返回这个 Bundle 使用的服务的列表，或者如果这个 Bundle 没有使用任何服务则返回 null。一个 Bundle 只有当其对服务的计数器大于 0 时被认为是正在使用一个服务。

如果 JRE 支持权限，只有调用者有使用服务注册时声明的至少一个类名称来获取这个服务的 ServicePermission 时，一个 ServiceReference 才被包含在列表中。

这个列表在调用这个方法的任何时候有效，然而，由于 Framework 是一个非常动态的环境，服务可能在任何时候被更改或卸载。

（2）返回值

一个 ServiceReference 对象数组或 null。

（3）异常

IllegalStateException——如果这个 Bundle 已经被卸载。

6.1.4.23 public int getState()

返回这个 Bundle 的当前状态。

一个 Bundle 在任何时刻只能处于一个状态：UNINSTALLED，INSTALLED，RESOLVED，STARTING，STOPPING，ACTIVE。

6.1.4.24 public String getSymbolicName()

（1）说明

返回这个 Bundle 的特征名称，这个名称在 Bundle-SymbolicName 清单头部指定。这个名称必须是独一无二的，它被建议为使用一个反向域名命名规范就像 Java 包使用的一样来命名。

这个方法将一直返回 Bundle 的特征名称直到 Bundle 处于 UNINSTALLED。

（2）返回值

这个 Bundle 的特征名称。

6.1.4.25 public boolean hasPermission(Object permission)

（1）参数

参数 permission：需要检验的权限。

（2）说明

校验这个 Bundle 是否有指定权限。

如果 JRE 不支持权限，这个方法总是返回 true。

权限是 Object 类型，以避免直接对 java.security.Permission 类的直接应用。这允许 Framework 可以在没有支持权限的 Java 运行时中实现。

如果 JRE 确实支持权限，这个 Bundle 和它的所有资源，包括内嵌的 JAR 文件，属于相同的 java.security.ProtectionDomain；也就是说，他们必须共享相同的权限集。

（3）返回值

如果这个 Bundle 有指定权限或者 Bundle 具有的权限包含了指定的权限，这返回 true；如果这个 Bundle 没有指定的权限或 permission 不是一个 java.security.Permission 实例，这返回 false。

（4）异常

IllegalStateException——如果这个 Bundle 已经被卸载了。

6.1.4.26 public Class loadClass(String name) throws ClassNotFoundException

（1）参数

参数 name：需要装载的类的名字。

（2）说明

使用这个 Bundle 的类加载器装载指定的类。

如果这个 Bundle 是一个片段 Bundle，那么这个方法必须抛出一个 ClassNotFoundException。

如果这个 Bundle 的状态是 INSTALLED，这个方法必须在尝试加载前去尝试解析。如果这个 Bundle 不能被解析，必须触发一个包含 BundleException 的 FrameworkEvent.ERROR 事件，这个 BundleException 包括有 Bundle 不能解析的详细原因，然后这个方法必须抛出一个 ClassNotFound。

如果这个 Bundle 的状态是 UNINSTALLED，那么一个 IllegalStateException 必须被抛出。

（3）返回值

返回值为请求的类对应的 Class 对象。

（4）异常

ClassNotFoundException——如果指定的类没有找到、如果这个 Bundle 是一个片段或者如果调用者没有合适的 AdminPermission[this,CLASS]且 JRE 支持权限，则抛出该异常。

IllegalStateException——如果这个 Bundle 已经被卸载，则抛出该异常。

6.1.4.27 public void start(int optons) throws BundleException

参数 options：启动这个 Bundle 的选项，查看 START_TRANSIENT 和 START_ACTIVATION_POLICY。Framework 必须忽略不认识的选项。

该方法用于启动 Bundle。

如果 Bundle 的状态是 UNINSTALLED，那么一个 IllegalStateException 将被抛出。

如果 Framework 实现了可选的启动级别服务而且当前启动级别低于这个 Bundle 的启动级别，则：

●如果 START_TRANSIENT 选项被设置，那么一个 BundleException 必须抛出，来指示这个 Bundle 由于 Framework 的启动级别原因而不能被启动。

●否则，框架必须设置 Bundle 的持久化 autostart 设置必须为：1）如果 START_ACTIVATION_POLICY 被设置，则为使用声明的激活设置启动；或 2）如果没有设置，则为使用立即激活启动。

当 Framework 当前启动级别变为等于或高于 Bundle 的启动级别，则这个 Bundle 将被启动。

（1） 如果这个 Bundle 处于正在激活或冻结过程，则这个方法必须等待激活或冻结过程结束。如果这没有在一个合理的时间内发生，则一个 BundleException 必须抛出用于指示这个 Bundle 不能够被启动。

（2） 如果这个 Bundle 的状态是 ACTIVE，那么这个方法立即返回。

（3） 如果 START_TRANSIENT 选项没有被设置，那么将根据选项设置持久化 autostart 设置（如果是 START_ACTIVATION_POLICY 选项，设置为使用声明激活策略启动，否则为立即激活启动）。

（4） 如果这个 Bundle 状态不是 RESOLVED，将尝试去解析这个 Bundle。如果 Framework 不能够解析这个 Bundle，则一个 BundleException 抛出。

（5） 如果设置了 START_ACTIVATION_POLICY 选项，而且这个 Bundle 声明的激活策略是晚激活，那么：如果这个 Bundle 的状态是 STARTING，则这个方法立即返回；这个 Bundle 的状态被设置为 STARTING；触发 BundleEvent.LAZY_ACTIVATION 事件；这个方法立即返回并且当 Bundle 激活被晚触发后，将完成接下来的步骤。

（6） Bundle 的状态设置为 STARTING。

（7） 触发 BundleEvent.STARTING 事件。

（8） 如果指定了 Bundle 激活器，则调用 BundleActivator.start 方法。如果激活器不可用或者抛出一个异常，那么：这个 Bundle 的状态设置为 STOPPING；抛出 BundleEvent.STOPPING 事件；卸载这个 Bundle 注册的服务；释放这个 Bundle 引用的服务；移除监听器；触发 BundleEvent.STOPPED 事件；抛出 BundleException。

（9） 如果这个 Bundle 的状态是 UNINSTALLED，因为这个 Bundle 在 BundleActivator.start 方法运行时这个 Bundle 被卸载了，那么一个 BundleException 将被抛出。

（10） 这个 Bundle 的状态设置为 ACTIVE。

（11） 触发 BundleEvent.STARTED 事件。

前置条件：

●getState()必须在{INSTALLED，RESOLVED}或如果这个 Bundle 有晚激活策略，则在{INSTALLED，RESOLVED，STARTING}。

没有异常抛出的后置条件：

●Bundle 的 autostart 设置被更改，除非设置 START_TRANSIENT 选项。

●getState()在{ACTIVE}，除非使用了晚激活策略。

●BundleActivator.start 方法被调用且没有抛出异常，除非使用晚激活策略。

当异常抛出的后置条件：

●依赖异常发生的时机，Bundle 的 autostart 设置将被更改除非 START_TRANSIENT 被设置。

●getState()不在{STARTING，ACTIVE }。

可能抛出的异常：

●BundleException——如果这个 Bundle 不能被启动，抛出这个异常。这可能由一个代码依赖不能被解析引起，或指定激活器不能被加载或抛出异常引起，也可能是这个 Bundle 是一个片段。

●IllegalStateException——如果这个 Bundle 已经被卸载且这个 Bundle 尝试改变它的状态时抛出。

●SecurityException——如果调用者没有合适的 AdminPermission[this,EXECUTE]且 JRE 支持权限时，抛出。

6.1.4.28 public void start() throws BundleException

不指定任何选项启动 Bundle。这个方法将调用 start(0)。

可能抛出的异常与 start(int optons)相同。

6.1.4.29 public void stop(int options) throws BundleException

参数 options：这个选项用于停止这个 Bundle。如 STOP_TRANSIENT。Framework 将忽略不认识的选项。

这个方法用于停止这个 Bundle。

以下步骤用于停止 Bundle：

（1） 如果这个 Bundle 状态是 UNINSTALLED，那么一个 IllegalStateException 将抛出。

（2） 如果这个 Bundle 处于正在激活或冻结状态，那么这个方法将等待激活或冻结完成。如果这个操作发生在不合适的时机，一个 BundleException 将被抛出来指示这个 Bundle 不能够被停止。

（3） 如果 STOP_TRANSIENT 选项没有被设置，那么这个 Bundle 的 autostart 持久化设置将设为 STOPPED。当 Framework 被重启后且这个 Bundle 的 autostart 设置是 STOPPED，这个 Bundle 将不能够被自动启动。

（4） 如果这个 Bundle 的状态不是 ACTIVE，那么这个方法将立即返回。

（5） 设置这个 Bundle 状态为 STOPPING。

（6） 触发 BundleEvent.STOPPING 事件。

（7） 如果指定了激活器，则调用 BundleActivator.stop 方法。如果这个方法抛出一个异常，将继续执行停止 Bundle 步骤，但在完成剩下的步骤之后，一个 BundleException 必须被抛出。

（8） 这个 Bundle 注册的服务必须被卸载。

（9） 这个 Bundle 使用的服务必须被释放。

（10） 这个 Bundle 注册的监听器必须被删除。

（11） 如果这个 Bundle 的状态是 UNINSTALLED，因为这个 Bundle 在调用 BundleActivator.stop 方法中被卸载了，那么必须抛出一个 BundleException。

（12） 这个 Bundle 的状态设置为 RESOLVED。

（13） 触发 BundleEvent.STOPPED 事件。

前置条件：

●getState()在{ACTIVE}。

没有异常发生的后置条件：

●除非设置 STOP_TRANSIENT 选项，那么 Bundle 的 autostart 设置将被更改。

●getState()不在{ACTIVE，STOPPING}。

●BundleActivator.stop 已经被调用而且没有抛出异常。

发生异常的后置条件：

●除非设置 STOP_TRANSIENT 选项，那么 Bundle 的 autostart 设置将被更改。

抛出的异常：

●BundleException——如果这个 BundleActivator 抛出一个异常或者这个 Bundle 是一个片段。

●IllegalStateException——如果这个 Bundle 已经被卸载了且尝试更改其状态。

●SecurityException——如果调用者没有 AdminPermission[this,EXECUTE]但 JRE 支持权限。

6.1.4.30 public void stop() throws BundleException

不指定选项来停止 Bundle。抛出的异常与 stop(int options)同。

6.1.4.31 public void uninstall() throws BundleException

（1）说明

卸载这个 Bundle。

这个方法会引起 Framework 通知其它 Bundle，这个 Bundle 正在被卸载，然后将这个 Bundle 的状态放入到 UNINSTALLED 状态。Framework 必须删除这个 Bundle 的能被删除的任何资源。

如果这个 Bundle 有导出任何包，Framework 必须使这些包对那些正在应用的 Bundle 可用直到 PackageAdmin.refresh 方法已经被调用或者 Framework 被重启。

卸载一个 Bundle 必须完成以下步骤：

1） 如果这个 Bundle 的状态是 UNINSTALLED 那么一个 IllegalStateException 将被抛出。

2） 如果这个 Bundle 的状态是 ACTIVE、STARTING 或 STOPPING，这个 Bundle 将以在 Bundle.stop 方法描述的来被停止。如果 Bundle.stop 抛出一个异常，一个包含这个异常的 FrameworkEvent.ERROR 事件将被触发。

3） 这个 Bundle 的状态设置为 UNINSTALLED。

4） 触发 BundleEvent.UNINSTALLED 事件。

5） 这个 Bundle 和任何由 Framework 提供给这个 Bundle 的持久存储区域将被删除。

（2）前置条件

●getState()没有在{UNINSTALLED}。

（3）没有异常的后置条件

●getState()在{UNINSTALLED}。

●这个 Bundle 已经被卸载。

（4）有异常的后置条件

●getState()没有在{UNINSTALLED}。

●这个 Bundle 没有被卸载。

（5）异常

BundleException——如果卸载失败者抛出。如果另一个线程正在尝试改变 Bundle 的状态且没有在合适的方式完成，可能抛出异常。

IllegalStateException——如果这个 Bundle 已经被卸载且这个 Bundle 尝试改变状态。

SecurityException——如果调用者没有合适的 AdminPermission[this,LIFECYCLE]且 JRE 支持权限。

6.1.4.32 public void update() throws BundleException

（1）说明

更新这个 Bundle。如果这个 Bundle 的状态是 ACTIVE，它必须在更新前停止并且在更新成功完成后启动。

如果这个 Bundle 导出任何包，这些包必须不能被更新。相反的，前一版本的包必须维持其导出知道 PackageAdmin.refreshPackages 方法被调用或者 Framework 被重启。

更新一个 Bundle 必须完成以下步骤：

1） 如果这个 Bundle 的状态是 UNINSTALLED，那么 IllegalStateException 将被抛出。

2） 如果这个 Bundle 的状态是 ACTIVE、STARTING 或 STOPPING，这个 Bundle 将首先按 Bundle.stop 方法描述的停止。如果 Bundle.stop 抛出一个异常，这个异常将被重新抛出终止更新。

3） 新版本的 Bundle 的位置决定于这个 Bundle 的 Constants.BUNDLE_UPDATELOCATION 清单头信息或 Bundle 的原始位置。

4） 这个位置依赖于具体实现来解释，电信的是一个 URL，且这个 Bundle 的新版本将从该位置获得。

5） 新版本 Bundle 被安装。如果 Framework 不能安装这个 Bundle 的新版本，这个 Bundle 的原始版本必须被恢复且在完成以下步骤后必须抛出一个 BundleException。

6） 如果这个 Bundle 声明了 Bundle-RequiredExecutionEnvironment 头信息，那么列表的执行环境必须和安装的执行环境对比。如果他们都不匹配，这个 Bundle 的原始版本必须被恢复且在完成以下步骤后必须抛出一个异常。

7） 这个 Bundle 状态设置为 INSTALLED。

8） 如果这个 Bundle 的新版本被安装，将触发 BundleEvent.UPDATED 事件。

9） 如果这个 Bundle 的原来状态是 ACTIVE，更新的 Bundle 必须以 Bundle.start 方法描述的方式启动。如果 Bundle.start 抛出一个异常，包含这个异常的 FrameworkEvent.ERROR 事件将被抛出。

（2）前置条件

●getState()没有在{UNINSTALLED}。

（3）没有异常的后置条件

●getState()在{INSTALLED，RESOLVED，ACTIVE}。

●这个 Bundle 已经被更新。

（4）有异常的后置条件

●getState()在{INSTALLED，RESOLVED，ACTIVE}。

●原始的 Bundle 仍被使用，没有更新发生。

（5）异常

BundleException——如果更新失败。

IllegalStateException——如果这个 Bundle 已经被更新且这个 Bundle 尝试更改自己的状态。

SecurityException——如果调用者没有合适的 AdminPermission[this,LIFECYCLE]且 JRE 支持权限。

6.1.4.33 public void update(InputStream in) throws BundleException

（1）参数

参数 in：读取新 Bundle 的 InputStream。

（2）说明

这个方法执行了 Bundle.update()的所有步骤，除了这个 Bundle 的新版本必须从提供的 InputStream 读取，而不是一个 URL。

这个方法必须当更新完成后关闭 InputStream，即使抛出一个异常。

（3）异常

BundleException——如果更新失败。

IllegalStateException——如果这个 Bundle 已经被更新且这个 Bundle 尝试更改自己的状态。

SecurityException——如果调用者没有合适的 AdminPermission[this,LIFECYCLE]且 JRE 支持权限。

6.1.5 public interface BundleActivator

自定义一个 Bundle 的启动和停止。

BundleActivator 是一个当 Bundle 启动或停止时被执行的接口。Framework 可以根据需要创建 BundleActivator 的实例。如果实例的 BundleActivator.start 方法执行成功，它将确保当 Bundle 被停止时调用相同实例的 BundleActivator.stop 方法。Framework 不能并行的调用一个 BundleActivator 对象。

BundleActivator 通过 Bundle-Activator 头信息指定。一个 Bundle 只能唯一指定一个 Bundle-Activator 头信息。一个 Bundle 在清单文件的 Bundle-Activator 头信息只能指定一个 BundleActivator。片段 Bundle 不能有 BundleActivator。清单头部的格式如下：

Bundle-Activator: class-name

其中 class-name 是一个 Java 类全名称。

指定的 BundleActivator 类必须有一个公共的无参构造器，这样一个 BundleActivator 对象可以通过 Class.newInstance()来创建。

6.1.5.1 public void start(BundleContext context) throws Exception

（1）参数

参数 context：这个正在启动的 Bundle 的执行上下文。

（2）说明

当 Bundle 被启动时调用，这样 Framework 可以执行启动这个 Bundle 需要的 Bundle 相关的活动。这个方法可以用于注册服务或分配这个 Bundle 需要的任何资源。

这个方法必须完成且以合适的方式返回调用者。

（3）异常

Exception——如果这个方法抛出一个异常，这个 Bundle 被标记为停止而且 Framework 将删除这个 Bundle 的监听器，卸载由这个 Bundle 注册的所有服务，释放由这个 Bundle 使用的所有服务。

6.1.5.2 public void stop(BundleContext context) throws Exception

（1）参数

参数 context：正在停止的 Bundle 的执行上下文。

（2）说明

当 Bundle 被停止时调用，这样 Framework 可以执行停止这个 Bundle 需要的 Bundle 相关的活动。一般的讲，这个方法需要撤销启动时 BundleActivator.start 的工作。当这个方法返回后，必须不存在由这个 Bundle 启动的任何未激活的线程。一个停止的 Bundle 不能调用任何 Framework 对象。

这个方法必须完成且以合适的方式返回调用者。

（3）异常

Exception——如果这个方法抛出一个异常，这个方法仍然被标记为停止，而且 Framework 将删除这个 Bundle 注册的监听器，卸载这个 Bundle 注册的所有服务，并释放这个 Bundle 使用的所有服务。

6.1.6 public interface Bundle

表示在 Framework 中一个 Bundle 的执行上下文。这个上下文用于授权访问关于 Frameowrk 对象的其它方法，这样这个 Bundle 可以与 Framework 交互。

BundleContext 方法必须允许一个 Bundle 完成：

- 订阅由 Framework 发布的事件。
- 使用 Framework 服务注册表注册服务对象。
- 从 Framework 服务注册表取出 ServiceReferences。
- 获取和释放一个引用服务的服务对象。
- 在 Framework 中安装新的服务。
- 获取 Framework 安装的 Bundle 列表。
- 为一个 Bundle 获取一个 Bundle 对象。
- 在框架提供的持久存储区的为文件创建文件对象。

当 Bundle 使用 BundleActivator.start 方法启动时，一个 BundleContext 对象将被创建并且使用这个上下文提供给关联的 Bundle。相同的 BundleContext 对象将被传递给这个上下文关联的 bundle，当使用 BundleActivator.stop 停止这个 Bundle 时。一个 BundleContext 对象通常是关联 Bundle 私有的而且并不与 OSGi 中的其它 Bundle 共享。

与一个 BundleContext 对象关联的 Bundle 对象称为上下文 Bundle。

BundleContext 只有在上下文 Bundle 执行期内有效，也就是说，只有在上下文 Bundle 处于 STARTING、STOPPING 和 ACTIVE 状态期内。如果 BundleContext 在执行期后使用，将抛出一个 IllegalStateException。当 Bundle 对象被停止后，BundleContext 对象必须不能重用。

Framework 是唯一能够创建 BundleContext 对象的实体而且上下文对象只有在 Framework 中创建才有效。

6.1.6.7 public Bundle getBundle()

（1）说明

返回与 BundleContext 关联的 Bundle 对象。这个 Bundle 称为上下文 Bundle。

（2）返回值

与 BundleContext 关联的 Bundle 对象。

（3）异常

IllegalStateException——如果 BundleContext 不再有效。

6.1.6.15 public Bundle installBundle(String location) throws BundleException

（1）参数

参数 location：Bundle 安装的位置标识。

（2）说明

从指定的位置字符串安装一个 Bundle。Framework 将位置字符串翻译为一个与实现无关的位置信息，从这个位置获得 bundle。

每一个安装的 Bundle 使用它的位置字符串唯一的标识，典型的格式为 URL。

以下是安装一个 Bundle 的步骤：

1） 如果一个包含相同位置字符串的 Bundle 已经被安装，那么返回这个 Bundle 的 Bundle 的对象。

2） 从位置字符串读取 Bundle 内容，如果读取失败，则抛出 BundleException 异常。

3） 这个 Bundle 的 Bundle-NativeCode 依赖将被解析。若失败，抛出 BundleException。

4） 分配 Bundle 关联的资源。最小的关联资源由一个唯一的标识，此外若平台有文件支持，它还包括一个持久存储区域。如果这个失败，抛出 BundleException。

5） 如果 Bundle 声明一个 Bundle-RequiredExecutionEnvironment 头信息，那么必将当前的执行环境与头信息列表的执行环境比较。如果没有一个列表的执行环境与当前安装的执行环境匹配，抛出 BundleException。

6） 设置 Bundle 的状态为 INSTALLED。

7） 触发 BundleEvent.INSTALLED 事件。

8） 返回新的 Bundle 或以前安装的 Bundle 的 Bundle 对象。

（3）没有异常的后置条件

●getState()在{INSTALLED，RESOLVED}。

●Bundle 有一个唯一的 ID。

（4）有异常的后置条件

●Bundle 没有被安装且没有存在这个 Bundle 的任何痕迹。

（5）返回值

安装的 Bundle 的 Bundle 对象。

（6）异常

BundleException——安装 Bundle 失败。

SecurityException——如果 JRE 支持权限系统但调用者没有合适的 AdminPermission[installed bundle,LIFECYCLIE]权限。

IllegalStateException——如果这个 BundleContext 不再有效。

7 包管理服务规范

7.1 介绍

Bundle 可以向其他 Bundle 导出包。这种导出创建了导出一个包的 Bundle 和使用这个包的 Bundle 的依赖。当导出包被卸载或更新，需要作出一个关于共享包的决策。

包管理服务提供一个接口使管理 Agent 来做决策。

7.1.1 要点

●信息（Information）——包管理服务必须提供所有包的共享状态。这应该包括引用 Bundle 和导出 Bundle 的信息。

●政策（Policy）——包管理服务必须允许一个管理 Agent 当 Bundle 被更新和卸载后提供一个包共享的政策。

●最小更新（Minimal update）——只有依赖需要解析的包的 Bundle 才应该在包被强迫刷新时被重启。

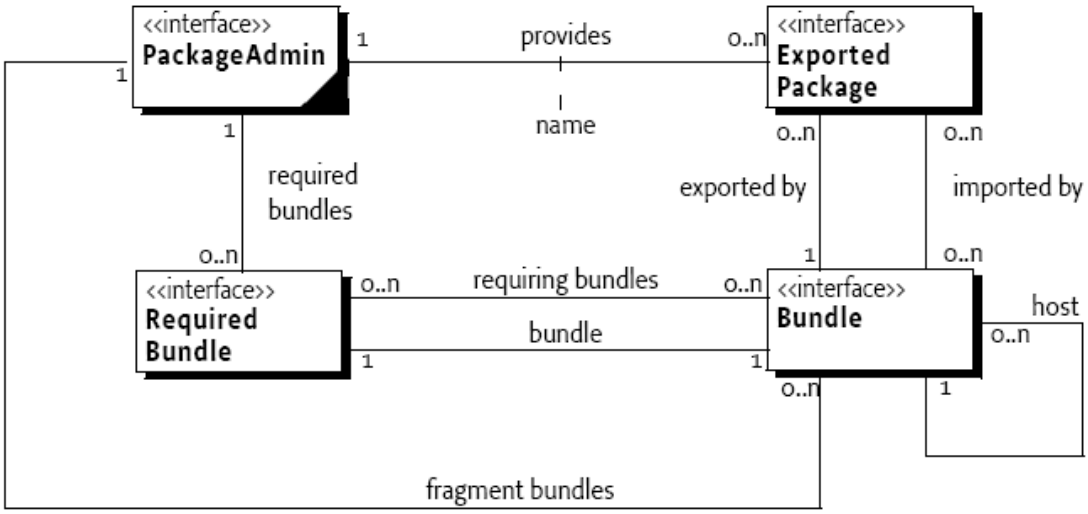
7.1.2 词汇

●PackageAdmin——提供访问内部框架包共享机制的接口。

●ExportedPackage——提供包信息和它的共享状态的接口。

●RequireBundle——提供关于必须包绑定的信息的接口。

●管理 Agent——由操作者提供实现一个操作者指定政策的一个 Bundle。



7.1.3 操作

框架的系统 Bundle 应该为管理 Agent 提供一个包管理服务。包管理服务必须由系统 Bundle 注册为 org.osgi.service.packageadmin.PackageAdmin 接口。它提供了访问框架关于包共享、片段和必须包的内部结构。这是一个可选的单例服务，因此在任何时刻必须至多注册一个包管理服务。

框架必须总是保证那些被卸载和更新的 Bundle 导出的包的包共享的完整性。一个管理 Agent 其后可以选择框架来使用包管理服务刷新这些包。一个总是使用当前最多由安装 Bundle 导出的包的政策可以被作为一个管理 Agent 实现，这个 Agent 会为那些正在被安装或更新的 Bundle 监视框架事件且然后使用包管理服务刷新那些 Bundle 的包。

7.2 包管理

包管理服务主要用于允许一个管理 Agent 来定义管理包共享的政策。它提供了检查共享包状态的方法。它也允许管理 Agent 来刷新包且根据需要停止和重启 Bundle。

7.2.1 包共享

类 PackageAdmin 提供了以下方法：

●getExportedPackage(String)——返回一个关于请求包的信息的 ExportedPackage 对象。这个信息可以用于做刷新包的决定。

●getExportedPackages(Bundle)——返回一个给定 Bundle 导出的每一个包的 ExportedPackage 对象列表。

●refreshPackages(Bundle[])——管理 Agent 可以调用这个方法来刷新指定 Bundle 的导出包。实际的操作必须是异步的。框架在所有包被刷新后必须发送一个 Framework.PACKAGES_REFRESHED 事件。

●resolveBundles(Bundle[])——框架必须尝试解析给定的 Bundle。

7.2.2 Bundle 信息

框架 API 中只有一个 Bundle 接口，然后 Bundle 在框架中可以执行不同的角色。包管理服务提供了这些结构信息的访问。

●getBundle(Class)——使用加载给定类的类加载器的相应 Bundle。

●getBundles(String,String)——使用给定 Bundle 特征名称和紧跟给定的版本查找 Bundle 的集合。如果版本为 null，所有使用给定特征名称的 Bundle 将被返回。

●getBundleType(Bundle)——符合的 Bundle 的类型。这是一个不同类型的位标识。以下标识被定义了：

A) BUNDLE_TYPE_FRAGMENT——这个 Bundle 是一个片段。

7.2.3 片段和必须 Bundle

包管理服务提供了由必须包和宿主片段创建的网络的访问。

●getFragments(Bundle)——返回作为给定 Bundle 的片段的 Bundle 数组。如果没有附加片段，则返回 null。

●getHosts(Bundle)——返回作为这个片段 Bundle 的宿主 Bundle。给定的 Bundle 必须是被附加的片段 Bundle，否则返回 null。

●getRequiredBundles(String)——返回匹配给定名称的 RequiredBundle 对象（或者如果名称为 null 则返回全部）。RequiredBundle 对象提供了关于一个必须 Bundle 的结构信息。

7.2.4 导出包

关于共享包的信息由 ExportedPackage 对象提供。这些对象提供了关于引用和导出包的详细信息。这些信息可以由管理 Agent 做决策时使用。

7.2.5 刷新包和启动级别服务

当 refreshPackages(Bundle[])方法被调用后，Bundle 可能会被停止且然后启动。如果启动级别服务存在，停止和启动 Bundle 必须不能违反启动级别的约束。这意味着使用更高的启动级别的 Bundle 必须在更低级别的 Bundle 停止前而停止。反之亦然，Bundle 不应该比其级别低的 Bundle 启动前启动。查看启动顺序一节。

7.3 安全

包管理服务是一个很容易被滥用的系统服务，因为它提供了框架的内部数据结构的访问。许多 Bundle 可能有 ServicePermission[org.osgi.service.packageadmin.PackageAdmin, Get]因为 AdminPermission[System Bundle, RESOLVE]需要来更改环境的任何方法。没有 Bundle 必须用于 ServicePermission[org.osgi.service.packageadmin.PackageAdmin, REGISTER]，因为只有框架本身应该注册这样的一个系统服务。

7.4

8 启动级别服务规范

这个规范描述了如何使管理 Agent 来控制关于 OSGi 服务平台中的 Bundle 的启动和停止顺序。

启动级别服务给每一个 Bundle 赋予一个启动级别。管理 Agent 可以为 Bundle 更改启动级别和设置框架的激活启动级别，这将会启动和停止合适的 Bundle。只有那些拥有低于或等于激活启动级别的启动级别的 Bundle 才必须被激活。

启动级别服务的目的是允许管理 Agent 来仔细的控制哪些 Bundle 将被启动与停止和这将在什么时候发生。

8.1 介绍

8.1.1 要点

●排序（Ordering）——一个管理 Agent 应该能够安排 Bundle 启动和停止的顺序。

●级别（Levels）——管理 Agent 应该支持一个无限制的数字级别。

●向后兼容（Backward compatible）——启动级别模型应该兼容 R2 规范。

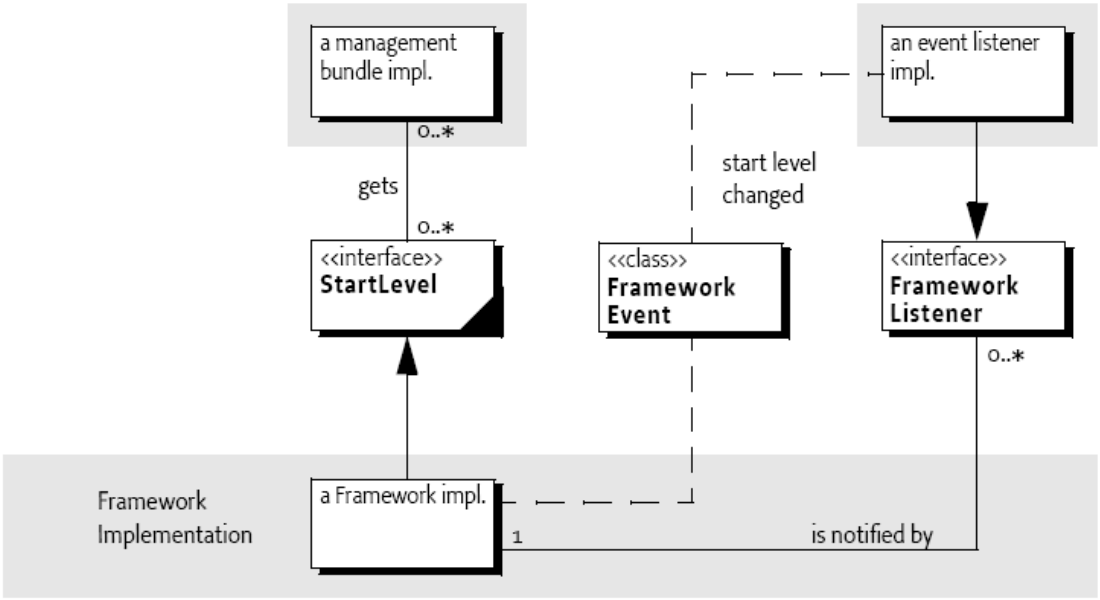
8.1.2 词汇

●启动级别服务——一个管理 Agent 使用这个服务来排列 Bundle 启动和停止的顺序。

●管理 Agent——查看第 32 页。

●框架事件——查看 97 页。

●框架监听器——查看 97 页。



8.2 启动级别服务

启动级别服务提供了一下功能：

●控制 OSGi 框架初始启动级别。

●用于更改框架的激活启动级别。

●可以拥有为一个 Bundle 赋予一个启动级别。

●可以设置新安装的 Bundle 的初始启动级别。

定义 Bundle 启动和停止的次序是有用的，这是因为：

●安全模式（Safe mode）——管理 Agent 可以实现一个安全模式。在这个模式，只有充分相信的 Bundle 被启动。当一个 Bundle 在启动时的失败导致了扰乱正常操作和阻止问题正确解析时安全模式可能是有效的。

●闪屏（Splash screen）——如果整个启动时间很长，它可能希望在初始化阶段显示一个闪屏。启动的排序可以确保正确的 Bundle 被最先启动。

●处理不稳定的 Bundle（Handling erratic bundles）——由于 Bundle 在他们被激活时需要一些服务是可能的，这将引起一些问题。通过控制启动顺序，管理 Agent 可以阻止这些问题。

●高级别 Bundle——一些任务，比如测量，需要尽可能的快速运行且不能有很长的启动延迟。这些 Bundle 必须首先被启动。

8.2.1 启动级别的概念

启动级别被定义为一个非负整数。0 启动级别是 Framework 没有启动或已经关闭的状态。在这个状态，没有运行任何的 Bundle。逐渐增加的更大的整数表示更高的启动级别。比如，2 是一个比 1 更高的启动级别。框架必须为启动级别支持所有的正整数。

框架有一个激活启动级别，它用于决定哪些 Bundle 可以被启动。所有的 Bundle 必须赋予一个 Bundle 启动级别。这是启动一个 Bundle 的最小级别。Bundle 启动级别可以通过 setBundleStartLevel(Bundle, int)方法来设置。当一个 Bundle 被安装后，它通过 getInitialBundleStartLevel()返回的启动级别来初始的设置 Bundle 启动级别。初始 Bundle 启动级别在安装时使用 setInitialBundleStartLevel(int)设置。

此外，一个 Bundle 可以通过 Bundle 的 start 和 stop 方法被持久的标记为启动或停止。不管 Bundle 的启动级别，一个 Bundle 除非它被标记为启动，否则不能被运行。